BlobSeer: Towards efficient data storage management for large-scale, distributed systems

Bogdan Nicolae

N^o d'ordre: 4310 **ANNÉE 2010**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention: INFORMATIQUE

Ecole doctorale MATISSE

présentée par

Bogdan Nicolae

préparée à l'unité de recherche n^o 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

BlobSeer: Towards efficient data storage management for large-scale, distributed systems

Thèse soutenue à Rennes le 30 novembre 2010

devant le jury composé de:

Luc BOUGÉ / directeur de thèse
Professeur, ENS Cachan Antenne de Bretagne, France

Gabriel ANTONIU / directeur de thèse
Chargé de recherche, INRIA Rennes, France

Franck CAPPELLO / rapporteur
Directeur de recherche, INRIA Saclay, France

Frédéric DESPREZ / rapporteur
Directeur de recherche, INRIA Grenoble, France

Kate KEAHEY / examinateur
Scientist, Argonne National Laboratory, USA

María PÉREZ / examinateur
Professor, Universidad Politécnica de Madrid, Spain

Valentin CRISTEA / examinateur
Professor, Politehnica University Bucharest, Romania

Jede Burg wurde Stein auf Stein aufgebaut.
– Siebenbürger Spruch

Every castle was build stone by stone.
– Transylvanian proveb

Acknowledgments

This dissertation was made possible through the patience and guidance of my advisors, Gabriel and Luc. I am most grateful for their constant support and encouragements that brought my work in the right direction and can only hope that it ends up as a solid foundation for many others.

Kind regards go to family: my parents, Adrian and Maria, as well as my sister Adriana and grandmother Maria. Their love and understanding has kept my motivation up throughout the duration of this doctorate and helped me overcome several difficult times.

I would also like to thank the members of the jury: Maria, Kate and Valentin for evaluating my work and traveling many miles to attend my defense. In particular many thanks to Kate for hosting me at Argonne National Laboratory, USA as a visiting student within the Nimbus project for a duration of two months. During this stay, I had the chance to apply the ideas of my work in a very challenging setting. Special thanks go to my two main evaluators, Franck and Frédéric, for taking their time to carefully read my manuscript and give me important feedback on my work.

Many thanks for many specific contributions from various people. Jesús Montes contributed with his expertise on global behavior modeling to refine the quality-of-service delivered by this work in the context of cloud storage. Matthieu Dorier helped during his internship with the integration of this work as a storage backend for Hadoop MapReduce, a joint work with Diana Moise. He was also kind enough to translate an extended 10-page abstract of this manuscript into French. Along Diana Moise, Alexandra-Carpen Amarie also contributed to a central publication around this work. Viet-Trung Tran applied this work in the context of distributed file systems, which led to a common publication. We also had many interesting exchanges of ideas towards the end of my doctorate and I sure hope we can purse them further beyond this work.

Thanks go as well to the various members of the KerData team: Houssem Chihoub and Radu Tudoran, as well as several people inside and outside of INRIA Rennes Bretagne-Atlantique: Eugen Feller, Eliana Tîrşa, Pierre Riteau, Alexandru Costan, Alice Mărăscu, Sînziana Mazilu, Tassadit Bouadi and Peter Linell. I happily recall the countless relaxing discussions at coffee-breaks and the quality time we spent together.

The experimental part of this work would not have been possible without the Grid'5000/ALADDIN-G5K testbed, an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners. Many thanks to Pascal Morillon and David Margery for their continuous support with Grid'5000.

Finally, many thanks to all other people that had a direct or indirect contribution to this work and were not explicitly mentioned above. Your help and support is very much appreciated.

Contents

Chapter 1

Introduction

Contents

We live in exponential times. Each year, 60% more information is generated in the world than in the previous year, with predictions that the total size of information will amount to 1800 Exabytes by the end of 2011. If we were to count the number of bits that represent information in circulation nowadays, we would already obtain a number that is higher than the estimated total number of stars in our entire universe.

Processing such vast amounts of data in order to infer new knowledge becomes increasingly difficult. Fortunately, computation, storage and communication technologies steadily improve and enable the development of complex data processing applications both in research institutions and industry. Since it is not feasible to solve such applications using a single computer, the idea arised to leverage the power of multiple autonomous computers that communicate through a computer network in order to solve them. Thus, the *parallel and distributed computing* research field emerged.

One particularly difficult challenge in this context is to find the right means to *store and manage* such huge amounts of data in a distributed environment. The main difficulty comes from the fact that in a distributed environment, *data needs to be shared* between autonomous entities such that they can converge towards a common goal and solve a problem. Data sharing is difficult, because the autonomous components need to agree how to manipulate the data such that it remains in a *consistent* state, yet try to perform as many data manipulations as possible in a *concurrent* fashion. Therefore, it is important to provide the necessary

abstractions that enable *high-performance data sharing at large scale*, otherwise the huge computational potential offered by large distributed systems is hindered by poor data sharing scalability. While this problem is well known, existing approaches still face many limitations that need to be overcome.

1.1 Objectives

Given the limitations of existing data storage approaches and new challenges that arise in the context of exponentially growing data sizes, this thesis aims at demonstrating that *it is possible to build a scalable, high-performance distributed data-storage service* that facilitates data sharing at large scale.

In order to achieve this main objective, this thesis aims to fulfill a series of sub-objectives:

1. To investigate and analyze a series of existing data storage approaches for distributed computing and to understand their limitations.
2. To formulate a series of design principles that enable the construction of a highly efficient distributed storage service.
3. To formalize the design principles into an algorithmic description that can be applied to implement such a distributed storage service.
4. To provide an efficient practical implementation of the storage service based on the algorithmic description.
5. To evaluate the implementation in a series of synthetic benchmarks that quantify the potential usefulness of the storage service
6. To adapt and evaluate the implementation in various applicative contexts that demonstrate its usefulness in concrete, real-life situations.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

Foundations for leveraging object-versioning to build scalable, distributed storage services. We propose a series of principles for designing highly scalable distributed storage systems which enable efficient exploitation of data-level parallelism and sustain a high throughput despite massively parallel data access. In particular, we defend versioning as a key principle that enhances data access concurrency, ultimately leading to better scalability and higher performance. We show how versioning makes it is possible to avoid synchronization between concurrent accesses, both at data and metadata level, which unlocks the potential to access data in a highly parallel fashion. This approach is combined with data striping and metadata decentralization, so that concurrent accesses are physically distributed at large scale among nodes.

BlobSeer: a high performance, large-scale distributed storage service based on these foundations. Based on the design principles mentioned in the previous paragraph, we introduce BlobSeer, a distributed storage service that was implemented and thoroughly tested in a series of experiments that validate the benefits of applying the proposed design principles. Our contribution introduces an architecture which is backed up by a series of algorithmic descriptions for manipulating objects under concurrency through versioning. In this context, we propose a segment tree-based metadata structure that enables efficient implementation of metadata forward references. We also present several techniques to integrate these contributions into a practical implementation, which we then evaluate extensively in a series of synthetic benchmarks that target various applicative contexts. This work was published in [110, 108, 104, 106, 107, 109, 160].

A BlobSeer-based storage layer that improves performance of MapReduce applications. MapReduce established itself as a prominent data-intensive computing paradigm in recent times. One of the core components of any MapReduce implementation is the underlying storage layer. In this context, we have designed and developed BlobSeer-based File System (BSFS), an efficient storage layer for Hadoop, an open-source MapReduce implementation. Our contribution consists in substituting the original storage layer of Hadoop (which is HDFS - Hadoop Distributed File System) with a new, concurrency-optimized data storage layer based BlobSeer, which enabled us to obtain significant performance improvement for data-intensive MapReduce applications. This improvement is confirmed through extensive large-scale experiments, both with synthetic benchmarks, as well as real-life MapReduce applications in common use. This work was carried out in collaboration with Diana Moise, Gabriel Antoniu, Luc Bougé and Matthieu Dorier. It was published in [112].

A series of techniques that leverage BlobSeer to improve virtual machine image deployment and snapshotting for IaaS clouds. In the context of an increasing popularity of cloud computing, efficient management of VM images such as concurrent image deployment to compute nodes and concurrent image snapshotting for checkpointing or migration are critical. The performance of these operations directly impacts the usability of the elastic features brought forward by cloud computing systems. Our contribution in this context is a lazy VM deployment scheme that leverages our versioning proposal to save incremental differences to persistent storage when a snapshot is needed, greatly reducing execution time, storage space and network traffic. Furthermore, the versioning principles of BlobSeer enable us to offer the illusion that each snapshot is a different, fully independent image. This has an important benefit in that it handles the management of incremental differences independently of the hypervisor, thus greatly improving the portability of VM images, and compensating for the lack of VM image format standardization. This work was carried out in collaboration with Kate Keahey and John Bresnahan at Argonne National Laboratory, Chicago, Illinois, USA, as well as Gabriel Antoniu, INRIA Rennes, France. It was published in [111].

A methodology to improve quality-of-service for cloud storage, illustrated on BlobSeer. The elastic nature of cloud computing model makes large-scale data-intensive applications highly affordable even for users with limited financial resources that cannot invest into expensive infrastructures necessary to run them. In this context, quality-of-service guarantees

are paramount: there is a need to sustain a stable throughput for each individual accesses, in addition to achieving a high aggregated throughput under concurrency. We contribute with a technique that addresses this need, based on component monitoring, application-side feedback and behavior pattern analysis to automatically infer useful knowledge about the causes of poor quality of service, and provide an easy way to reason about potential improvements. This technique is applied to BlobSeer and thoroughly tested in a series of representative scenarios, where it demonstrated substantial improvements in the stability of individual data read accesses under MapReduce workloads. This work was carried out in collaboration with Jesús Montes and María Pérez from Universidad Politécnica de Madrid, Spain, with Alberto Sánchez from Universidad Rey Juan Carlos, Madrid, Spain and with Gabriel Antoniu, INRIA Rennes, France. It was published in [98].

All experiments involved in the aforementioned contributions were carried out on the Grid'5000/ALLADIN experimental testbed federating 9 different sites in France. It is an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners. We are particularly grateful for the excellent support that was provided by the Grid'5000 team during the time in which the work presented in this thesis was carried out.

1.3 Publications

The work presented in this manuscript was published in several peer-reviewed venues and research reports. They concern:

- the core principles of BlobSeer and their algorithmic implementation [110, 108, 104, 106];
- the potential benefits of BlobSeer for scientific applications [107] and Desktop Grids [109] using synthetic benchmarks;
- MapReduce applications, where BlobSeer demonstrated significant performance gains over standard storage services [112];
- the advantages of BlobSeer as a cloud storage service that efficiently manages virtual machine images [111], offers high quality-of-service guarantees [98], and offers high throughput compression [105];
- the advantages of BlobSeer as a building block for grid file systems [160].

Journal articles

- Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise and Alexandra Carpen-Amarie. **BlobSeer: Next Generation Data Management for Large Scale Infrastructures.** In *Journal of Parallel and Distributed Computing*, 2010, In press.

Conferences

- Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé and Matthieu Dorier. **BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map/Reduce Applications.** In *IPDPS '10: Proc. 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1-12, Atlanta, USA, 2010.

- Jesús Montes, Bogdan Nicolae, Gabriel Antoniu, Alberto Sánchez and María Pérez. **Using Global Behavior Modeling to Improve QoS in Data Storage Services on the Cloud.** In *CloudCom '10: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010, In press.

- Bogdan Nicolae. **High Throughput Data-Compression for Cloud Storage.** In *Globe '10: Proc. 3rd International Conference on Data Management in Grid and P2P Systems*, pages 1-12, Bilbao, Spain, 2010.

- Bogdan Nicolae. **BlobSeer: Efficient Data Management for Data-Intensive Applications Distributed at Large-Scale.** In *IPDPS '10: Proc. 24th IEEE International Symposium on Parallel and Distributed Processing: Workshops and Phd Forum*, pages 1-4, Atlanta, USA, 2010, Best Poster Award.

- Bogdan Nicolae, Gabriel Antoniu and Luc Bougé. **Enabling High Data Throughput in Desktop Grids Through Decentralized Data and Metadata Management: The BlobSeer Approach.** In *Euro-Par '09: Proc. 15th International Euro-Par Conference on Parallel Processing*, pages 404-416, Delft, The Netherlands, 2009.

- Viet Trung-Tran, Gabriel Antoniu, Bogdan Nicolae, Luc Bougé and Osamu Tatebe. **Towards A Grid File System Based On A Large-Scale BLOB Management Service.** In *EuroPar '09: CoreGRID ERCIM Working Group Workshop on Grids, P2P and Service computing*, pages 7-19, Delft, The Netherlands, 2009.

- Bogdan Nicolae, Gabriel Antoniu and Luc Bougé. **BlobSeer: How to Enable Efficient Versioning for Large Object Storage under Heavy Access Concurrency.** In *Proc. EDBT/ICDT '09 Workshops*, pages 18-25, St. Petersburg, Russia, 2009.

- Bogdan Nicolae, Gabriel Antoniu and Luc Bougé. **Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection.** In *Cluster '08: Proc. IEEE International Conference on Cluster Computing: Poster Session*, pages 310-315, Tsukuba, Japan, 2008.

- Bogdan Nicolae, Gabriel Antoniu and Luc Bougé. **Distributed Management of Massive Data: An Efficient Fine-Grain Data Access Scheme.** In *VECPAR '08: Proc. 8th International Meeting on High Performance Computing for Computational Science*, pages 532-543, Toulouse, France, 2008.

Research reports

- Bogdan Nicolae, John Bresnahan, Kate Keahey and Gabriel Antoniu. **Going Back and Forth: Efficient VM Image Deployment and Snapshotting** *INRIA Research Report No. 7482*, INRIA, Rennes, France, 2010.

1.4 Organization of the manuscript

The rest of this work is organized in four parts.

The first part: introduces the context of our work, presenting the state of the art in the related research areas. It consists of Chapters 2 and 3. Chapter 2 presents a high-level overview on distributed computing paradigms that are designed to scale to large sizes, in order to solve complex problems that require large amounts of computational power and manipulate massive amounts of data. In particular, we focus on clusters, grids and clouds. Chapter 3 narrows the focus on data storage and management. After identifying the main properties that data storage should fulfill in a distributed environment, several existing approaches designed for clusters, grids and clouds are analyzed with respect to those properties. We conclude with a discussion about the limitations of existing approaches and new challenges that data storage faces in the light of the ever growing scales of distributed systems.

The second part: introduces the core contribution of this work: BlobSeer, a distributed data storage service that aims at addressing several of the challenges that were discussed in the first part. In it organized in four chapters. Chapter 4 proposes a series of general design principles that we consider to be crucial for overcoming the aforementioned challenges. In particular, we insist on the importance of versioning as a key to enhancing data access concurrency, ultimately leading to better scalability and higher performance. Chapter 5 presents the architecture of BlobSeer and gives a high-level overview on how the basic data manipulation primitives work. It then introduces an algorithmic description of the versioning principles presented in Chapter 4. Chapter 6 focuses on the metadata management in BlobSeer. In particular, we introduce the algorithmic description of a highly-scalable distributed metadata management scheme for our versioning algorithms that is specifically designed to improve metadata accesses under heavy concurrency. Chapter 7 discusses the BlobSeer implementation in real-life, insisting on software engineering aspects and other practical issues and technical details that we encountered. Finally, Chapter 8 evaluates the implementation described in Chapter 7 through a series of synthetic benchmarks that consist of specific scenarios, each of which focuses on the design principles presented in Chapter 4.

The third part: presents a series of contributions that leverage BlobSeer in the context of several real-life applications, demonstrating the potentially large benefits of our proposal. It is organized in 3 chapters. Chapter 9 evaluates BlobSeer in the context of MapReduce applications, for which we designed and implemented a layer on top of BlobSeer that provides a specialized MapReduce file system API. We compare our approach to the Hadoop Distributed File System, which is the default storage solution of Hadoop, a popular open-source MapReduce framework, and show significant improvement. Chapter 10 proposes a storage platform built on top of BlobSeer that optimizes virtual machine image manipulations on clouds. In particular, we address the issues of efficiently deploying multiple virtual machines at the same time, as well as efficiently snapshotting virtual machines simultaneously to persistent storage. We show significant speedup and lower resource consumption of our approach compared to more traditional approaches. Finally, Chapter 11 proposes a gen-

eral methodology to improve quality-of-service for cloud storage based on global behavioral modeling. We experiment with synthetic MapReduce access patterns and show significant reduction in throughput variability under concurrency.

The fourth part: is represented by Chapter 12 and summarizes the aforementioned contributions, discusses the limitations of our work and presents a series of future perspectives that are interesting to explore.

Part I

Context: data storage in large-scale, distributed systems

Chapter 2

Large scale, distributed computing

Contents

As information grows at an exponential rate [52], so does the complexity of applications that need to manipulate this information in order to infer new knowledge. A single computer, no matter how powerful, cannot keep up with this trend. Therefore, a natural idea that emerged in this context was to leverage the power of multiple autonomous computers that communicate through a computer network in order to achieve a common goal. An infrastructure that implements this idea is called a *distributed system* [6].

The drive for larger and faster distributed systems that aggregate the power of more and more computers triggered a rapid evolution of research in this direction.

Distributed computing started out of necessity to solve mission-critical problems, such as simulations of natural events for the purpose of predicting and minimizing disasters. Such applications are usually tightly coupled and typically need large amounts of computing power for short periods of time (i.e. days or weeks) to answer questions like "where

will the hurricane strike?". High performance is critical in this context: the answer is needed as fast as possible. To address this need, supercomputers are build that leverage the latest of computing and network infrastructure, but are difficult and expensive to maintain (high energy consumption, complex cooling systems, etc.). High costs saw the use of supercomputers solely at national and international public institutions that can afford to pay for them. The field that studies this type of distributed systems is called *high performance computing (HPC)*.

With increasing application complexity, eventually even smaller institutions and private companies adopted distributed computing to run their every-day applications. In this context, the main driving factor is money: how to get as much computational power for the lowest price. Therefore, efficiency is not measured in performance delivered over short amounts of time, as is the case with HPC, but rather throughput: how many applications can be run over the course of months or even years to amortize the infrastructure costs. Applications are typically coarse-grain and perform simple computations (i.e., embarrassingly parallel). To address this need, distributed systems are built out of loosely-coupled commodity hardware that is much cheaper to buy and maintain than supercomputers. Such systems are the object of *high throughput computing (HTC)*.

With the explosion of data sizes, applications shifted from being computationally intensive to data-intensive. They can usually be formulated as embarrassingly parallel problems that involve a filtering or funneling process. More precisely, they start with vast amounts of data and end with simple answers, often as small as one-bit decisions: yes or no, buy or sell, etc. This requires taking vast amounts of unstructured raw data through a series of processing steps that refine it to become more comprehensive and include better insight, ultimately leading to better decision making. This type of applications prompted the introduction of huge distributed systems both in the public and private sector that challenge even the most powerful supercomputers. Such systems specialize to deliver a high data processing throughput and are studied by *data-intensive computing* [24, 119].

A clear line between high performance computing, high throughput computing and data-intensive computing cannot be drawn. All evolved together and influenced each other. In the drive to lower costs, recent trends try to abridge the gap between them. For example, *many-task computing (MTC)* [124] tries to adopt the cost-effective principles of high throughput computing to solve high performance computing problems. This chapter focuses mainly on high throughput computing and data-intensive computing, presenting the evolution of distributed systems from *clusters* to *grids* and finally *clouds*.

2.1 Clusters

Clusters emerged as a first effort to assemble commodity hardware in order to build inexpensive distributed systems. They typically consist of personal computers and/or workstations (called *nodes*) that are linked through basic networking infrastructure, such as Ethernet. The simplicity of this approach, coupled with low entry and maintenance cost, made clusters highly popular. Even nowadays, clusters are adopted in all possible scales: from a couple of nodes to tens of thousands.

2.1.1 Computing clusters

Computing clusters aim to provide scalable solutions that can handle the increasing complexity of applications, both in size and scope.

A first effort in this direction was *Beowulf* [94, 93], originally referring to a specific cluster build at NASA out of commodity hardware to emulate a supercomputer. The term was later extended to include a whole class of clusters that run a standardized software stack: GNU/Linux as the operating system and *Message Passing Interface (MPI)* or *Parallel Virtual Machine (PVM)* on top of it [25], with the aim of providing a cost-effective and portable alternative to supercomputers.

A significantly different approach was undertaken by *Condor* [157], a middleware that coined the term high throughput computing. Rather than trying to emulate a supercomputer that is able to run tightly-coupled, computationally-intensive applications, its goal is to enable coarse-grained parallelization of computationally-intensive applications. Condor can both leverage dedicated clusters of computers and/or the idle CPU cycles of regular desktop machines when they are not in use. Nowadays, dedicated Condor clusters are widely used even by public institutions (such as NASA) and reach thousands of nodes.

The need to process massive data sizes by industry giants such as Google and Yahoo prompted the introduction of huge clusters made out of commodity parts that minimize per unit cost and favor low power over maximum speed. Google for example does not disclose the size of their infrastructure, but it is widely believed [89] it amounts to several million processors spread in at least 25 data centers, which are grouped in clusters of tens of thousands. Disk storage is attached to each processor to cope with the vast data sizes, while processors are interconnected with standard Ethernet links.

2.1.2 Load-balancing clusters

Load-balancing clusters link together multiple computers with the purpose of providing the illusion of a single powerful machine, called *single system image (SSI)*. Unlike other systems that typically operate at job-level, a SSI operates at process level: processes started by users appear to run locally but are transparently migrated to other nodes in order to achieve load balancing.

MOSIX [13, 56] was one of the first SSI implementations, incorporating automatic resource discovery and dynamic workload distribution, commonly found on single computers with multiple processors. Kerrighed [102, 86] is another SSI that builds on the same principles as MOSIX but introduces several advanced features such as support for cluster wide shared memory and transparent process checkpointing.

Load-balancing clusters are typically used for applications that need lots of RAM and processing power, such as graphical rendering, compilation of large repositories and online gaming. They comprise a small number of nodes, in the range of tens to hundreds.

2.1.3 High-availability clusters

Finally, a basic use of clusters is to provide high availability services. In order to do so, data is replicated and cached on multiple nodes, which enables a certain degree of load-balancing

and redundancy, effectively avoiding a single server to act as a bottleneck and single point of failure. Normally, if a server hosting a particular application crashes, the application becomes unavailable until the system administrator fixes it. With high availability clusters, hardware and software faults are automatically detected and fixed, without human intervention. This is called *fail-over* and can implemented at different levels, from very low such as simple redirection of network traffic to a different server to complex schemes implemented at application level.

High availability clusters are often used for critical databases, file sharing on a network, business applications, and customer services such as commercial websites. Their size is typically very small, in the order of tens of nodes, often numbering as little as two nodes, since it is the minimum required to provide redundancy.

2.2 Grids

Clusters proved to be a very powerful tool, and is widely adopted by many organizations. A natural question that arises in this context is whether it is possible to federate the resources of multiple organizations in order to obtain even more powerful distributed systems. *Grids* are concerned with precisely this question, proposing a solution that enables taking advantage of resources distributed over wide-area networks in order to solve large-scale distributed applications.

The term *grid* was first defined in [48] as "a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities". It originates from an analogy with the electrical power grid: the distributed system should provide computational power to any user of the grid at any moment in a standard fashion, as easy as plugging an electrical appliance into an outlet. This generic definition has been used in a lot of contexts to the point where it became difficult to understand what a grid really is. In [49], Foster, Kesselman, and Tuecke try to refine the grid definition to a distributed system that enables "coordinated resource sharing and problem solving in dynamic, multi-institutional, virtual organizations".

The concept of *virtual organization (VO)* is central in grids. The premise is that the grid is formed from a number of mutually distrustful participants form a consortium, with the purpose of sharing resources to perform a task. Sharing in this context refers to complex interactions, such as direct access to remote software, computers, data, etc. These interactions are enforced in a highly controlled fashion: resource providers and consumers clearly state under what conditions who is allowed to share what. A set of participants defined by such sharing rules is called a virtual organization.

The grid is thus far from being a "well-behaved" distributed system. As pointed out in [28], assumptions such as rare failures, minimal security, consistent software packages and simple sharing policies that work very well for clusters cannot be relied upon in grids. In [46], Foster proposes a three-point checklist of requirements that any grid should meet:

1. **Coordinates resources that are not subject to centralized control.** The grid coordinates resources that belong different administrative domains and as such it must address issues of security, policy enforcement, access control, etc. that arise in this context.

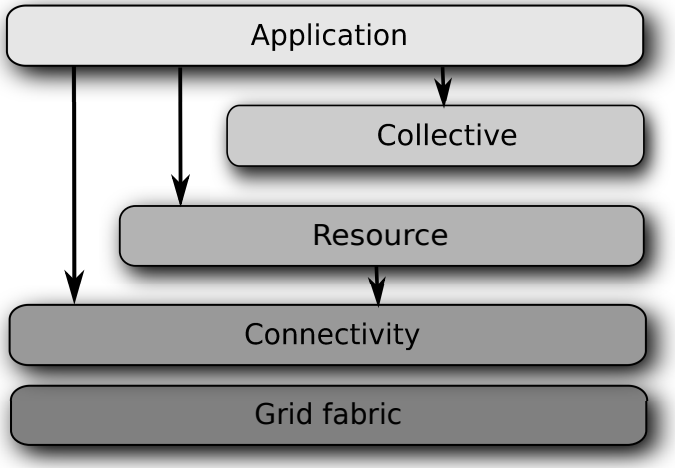


Figure 2.1: Generic architecture of the grid

Since each members are distrustful of each other, these issues cannot be addressed in a centralized fashion.

2. **Using standard, open, general-purpose protocols and interfaces.** Resource sharing is relies on multi-purpose protocols and interfaces that address issues such as authentication, authorization, resource discover, resource access, etc. Using standards is crucial in this context, as it facilitates interactions between the members and can be used to form reusable building blocks that work for a large number of applications.

3. **To deliver nontrivial qualities of service.** Since grids distribute applications over large geographical areas, they must be able to deliver various qualities of service such as response time, throughput, availability, security, co-allocation of multiple resource types to meet complex user demands, etc., so that it becomes a distributed system that is more powerful than the sum of its parts.

2.2.1 Architecture

A generic architecture for grids, that places few constraints on design and implementation was proposed in [49] and is show in figure 2.1.

The *grid fabric* provides the lowest access level to raw resources that make up the grid (clusters of computers, individual computers, file servers, etc.). It implements a unified interface for resource monitoring and management through a series of drivers that are adapted to a large number of native systems. The *connectivity* layer is responsible to enable communication between the grid resources, addressing issues such as authentication and security. The *resource* layer builds on top of the connectivity layer in order to implement the protocols that expose the individual resources to the grid participants. It provides two important functionalities to the upper layers: the ability to query the state of a resource and the mechanism to negotiate access to a resource. The *collective* layer builds on both the connectivity layer and resource layer to coordinate individual resources. It is responsible to provide functionalities such as resource discovery, scheduling, co-allocation, etc. Finally, the *application* layer makes

use of all other layers to enable the implementation of applications at virtual organization level.

2.2.2 Middleware

Several corporations, professional groups, university consortiums, and other groups are involved with the development of middleware that facilitates the creation and management of grids.

Globus. The Globus Toolkit [47] is a de-facto standard grid computing middleware adopted both in the academia and industry: HP, Cray, Sun Microsystems, IBM, etc. It is an open-source and open-architecture project governed by the *Globus Alliance*, an organization that aims at standardizing protocols and basic services for constructing grids. Globus implements solutions for security, resource management, data management, communication and fault tolerance. It is designed in a modular fashion that enables each participant in the grid to selectively enable the functionality it desires to expose to other participants without breaking overall compatibility.

UNICORE. *UNiform Interface to COmputing REsources (UNICORE)* [133] is a middleware developed in the context of two projects funded by the German ministry for education and research (BMBF) and has matured to the point where it is used in several production grids and many European and international research project, such as EUROGRID, GRIP, OpenMolGRID, VIOLA, NaReGI, etc. UNICORE implements a Graphical User Interface (GUI) that enables intuitive, seamless and secure access to the underlying services. The underlying services rely on *Abstract Job Objects (AJO)*, which are the foundation of UNICORE's job model. An AJO contains platform and site independent descriptions of computational and data related tasks, resource information and workflow specifications. AJOs are a flexible tool that enables building complex applications that are bound to many constraints and interactions.

gLite. The *gLite* [88] middleware, developed in the context of the EGEE [78] project is the foundation of many large scale scientific grids. CERN for example adopted gLite for the Worldwide LHC Computing Grid (WLCG). Initially based on the Globus toolkit, gLite evolved independently into a completely different middleware that targets production grids, aiming to improve usability. To this end, a rich user interface is provided that enables a variety of management tasks, such as listing all the resources suitable to execute a given job, submitting/canceling jobs, retrieving the output of jobs, retrieving logging information about jobs, upload/replicate/delete files from the grid, etc.

2.3 Clouds

In theory, the grid has a high potential to achieve a massive aggregated computational power, provided a large number of participants are willing to share their resources for a common goal. A large number of participants however introduces several difficulties in

practice. Since each member is responsible for its own resources and can enter or leave the consortium at any time, grids become highly dynamic in nature and make quality-of-service difficult to achieve. Furthermore, virtual organizations introduce complex security and management policies that are not easy to handle. Both users and application developers often feel that "there is too much to configure" in order to get started on the grid.

Clouds [168, 163] emerged as a paradigm that evolved from grids with the promise to provide reliable and user-friendly services delivered through next-generation data centers that are build on virtualized computational and storage technologies. Much like grids, a cloud federates computational resources into a single entity that enables its users to leverage computational power to solve a problem in the same way they can plug in an appliance into an outlet, without having to care where the electricity comes from. However, unlike grids, clouds are driven by an *economy model* rather than the need to form a consortium in which resources are shared. More specifically, clouds are owned by service providers that let consumers utilize cloud resources in a pay-as-you-go fashion: the consumer pays only for the resources that were actually used to solve its problem (for example: bandwidth, storage space, CPU utilization).

In particular, consumers indicate the required service level through quality-of-service parameters, which are noted in contracts, called *service level agreements (SLAs)*, that are established with the providers. Consumers are guaranteed that they will be able to access applications and data from the cloud anywhere in the world on demand. Moreover, guarantees are given that the cloud is robust and highly available. In [23], Buyya et al. propose the following definition: "A cloud is a type of parallel and distributed system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resource(s) based on service-level agreements established through negotiation between the service provider and consumers."

The economy-driven model adopted by clouds has a series of interesting advantages:

Low entry and maintenance costs. Clouds convert large computation costs from capital expenditures to operational expenditures. This enables consumers that do not have the budget or do not want to buy and maintain their own infrastructure (for example, small companies or start-ups that need one-time or infrequent computations) to still be able to run their desired applications. As noted in [81], clouds lead to dynamic and competitive market offers that are predicted to lower overall costs as they mature.

Elasticity. Since resources are provisioned on demand, consumers can dynamically upscale or downscale their applications to fit their needs. This flexibility avoids the situation when consumers are forced to buy expensive hardware to deal with peak data processing times, only to see that hardware under-utilized otherwise.

Scalability. Since it is in the interest of the providers to serve as many customers as possible, clouds can easily grow to huge sizes. Thus, a consumer is able to utilize virtually an unlimited number of resources, provided it has the money to pay for them.

Rapid development. By using clouds, consumers do not have to go through a lenghtly process of buying and setting up their infrastructure in order to run their applications. All details of hardware and software configuration and maintenance are handled by the cloud provider, which enables the consumer to focus on the application only.

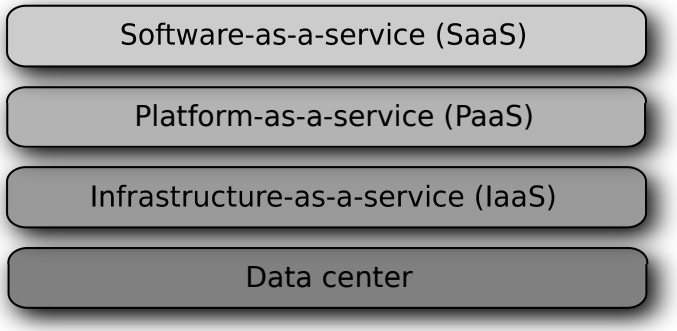


Figure 2.2: Cloud services and technologies as a stack of layers

2.3.1 Architecture

Cloud technologies and services can be classified into a stack of layers [79], as illustrated in figure 2.2:

Data center. The data center layer consists of the hardware and software stack on top of which the cloud services are build, including clusters of computers, networking infrastructure, operating systems, virtualization technologies, etc.

Infrastructure-as-a-Service (IaaS). IaaS typically offers raw computation and storage solutions in form of a virtualization environment and distributed storage service respectively. Rather than directly renting servers, software, disks or networking equipment, cloud consumers customize virtual machine images, store the images and application data remotely using the storage service, and then finally launch multiple VM instances on the cloud. Fees are charged on an utility basis that reflects the amount of raw resources used: storage space-hour, bandwidth, aggregated cpu cycles consumed, etc. A popular cloud service provider is Amazon, with its offer Amazon EC2 [130].

Platform-as-a-Service (PaaS). Moving up in the hierarchy, PaaS builds on IaaS to provide higher level programming and execution environments. Services at this level aim at freeing the consumer from having to configure and manage industry-standard application frameworks (for example Hadoop [169]), on top of which distributed applications are build, directly at IaaS level.

Software-as-a-Service (SaaS). At the highest level is SaaS, which aims at delivering end-user applications directly as a service over the Internet, freeing the consumer from having to install any software on its own computer or care about updates and patches. Most often, a simple web browser is enough to perform all necessary interaction with the application. SaaS is becoming increasingly popular, with industry giants such as Google advocating for light-weight operating systems that eliminate the need to install user applications altogether.

2.3.2 Emerging platforms

The economy model behind clouds prompted their adoption especially in the private sector. Industry giants such as: Amazon, Google, IBM, Microsoft, etc. develop and offer a wide range of cloud services. At the same time, cloud projects are also under development in academia as a series of research projects and open source initiatives [29].

Amazon EC2. *EC2* [130] provides a virtual computing environment that exposes a web service interface to the consumer through which it can launch virtual instances of a variety of operating systems, that can be loaded with custom application environments. The consumer can dynamically adjust the number of such instances through the same interface. A large pool of predefined virtual machine images, called *Amazon Machine Images (AMIs)* is provided, that can be directly used as such or customized to form new AMIs. The cost for using EC2 is measured in instance-hours. A specialized storage service, *Amazon S3* [130], is provided that is responsible to store both AMIs and consumer data. This service charges for amount of data transfers and GB-hour.

Google App Engine. *App Engine* [139] is a PaaS that enables consumers to to build and host web apps on the same systems that power Google applications. It offers fast development and deployment that is coordinated through simple, centralized administration. Targeted at casual users, it is free up to a certain resource utilization level, after which a low pricing scheme is applied. Fees are charged for storage space-hour, bandwidth and CPU cycles required by the application.

Microsoft Azure. *Azure* [126] is the cloud offer from Microsoft that runs on a large number of machines, all located in Microsoft data centers. It is based on a fabric layer that aggregates the computational resources into a whole, which is the used to build compute and storage services that are offered to the consumer. Developers can build applications on top of languages commonly supported by Windows, such as C#, Visual Basic, C++, Java, ASP.NET, using Visual Studio or another development tool.

Nimbus. *Nimbus* [71] is an open source toolkit that allows institutions to turn their cluster into an IaaS cloud. It it interface-wise compatible with the Amazon EC2 API [130] and Grid community WSRF. Data storage support is provided by *Cumulus*, which is compatible with the Amazon S3 API. Internally, Nimbus can rely both on Xen and KVM as virtualization technologies and can be configured to use standard schedulers for virtual machine deployment such as PBS and SGE. It is based on an extensible architecture that allows easy customization of provider needs.

OpenNebula. OpenNebula [101] is another open-source toolkit, specifically designed to support building clouds in any combination: private, public and hybrid. It can be integrated with a wide range of storage and networking solutions to fit a broad class of data centers, in order to form a flexible virtual infrastructure which dynamically adapts to changing workloads. An interesting feature of OpenNebula is its ability to combine both private-owned

data center resources with remote cloud resources, which gives providers greater flexibility by allowing them to act as consumers themselves.

Eucalyptus. Eucalyptus [113] is an open-source toolkit that started as an NSF funded research project at University of California, Santa Barbara. It implements IaaS using existing Linux-based infrastructure found in modern data centers. Its interface is compatible with Amazon's EC2 API [130] enabling movement of workloads between EC2 and data centers without modifying any code. Internally, Eucalyptus can rely on several virtualization technologies, including VMware, Xen, and KVM.

2.4 Conclusions

In our present-day, dynamic society it becomes increasingly difficult to keep up with the explosion of information. For this reason, distributed computing systems were introduced as a solution that helps processing such huge amounts of information in order to infer new knowledge out of it. This chapter presented the evolution of distributed systems, from clusters to grids and finally clouds. With modern datacenters hosting tens of thousands of nodes, distributed systems have a huge computational potential.

However, in order for this potential to be leveraged at its maximum, distributed systems must be designed in such way that they are able to store and manage huge amounts of data in an efficient fashion. This aspect is the focus of the next chapter.

Chapter 3

Data storage in large-scale, distributed systems

Contents

DATA storage and management plays a crucial role in leveraging the computational potential of distributed systems efficiently. This aspect forms the focus of this chapter.

In order to enable efficient and reliable access to data, several important design issues need to be taken into consideration:

High performance. A crucial aspect of data storage is the performance of data accesses. Since every application needs to process input data and generate output data, how fast data accesses can be executed impacts the total execution time of the application as a whole. This issue is especially important in the context of data-intensive computing, where data accesses represent a large portion of the application.

Scalability. Since there is a need to build larger and larger distributed systems, it is crucial to keep the same level of performance for data accesses when the number of clients that are concurrently served by the storage system increases.

Data access transparency. With distributed systems growing in size, it becomes increasingly difficult for applications to manage the location of data and move data from one location to another explicitly. *Transparency* is an important feature that addresses this problem. Rather than managing data locations explicitly, applications use a global namespace and a uniform access interface that enables data to be accessed in the same fashion, regardless of data location. Support for transparency greatly simplifies application development, enabling for example migration of processes without changing the way data is accessed.

Versioning support. With the growing amount of data that needs to be stored, it becomes increasingly important to provide support for versioning, i.e., to keep track of how data changes throughout time and enable the user to retrieve data from any past point in time. For example, in many cases it is necessary to undo updates to the data that happened by accident. Versioning is also enforced in many cases by legislation: institutions are often required to keep an auditable trail of changes made to electronic records, which is a complex issue to manage at application level without versioning support at the level of the storage system.

Concurrency control. Scalability can only be achieved if the storage system enables its clients to access data concurrently. However, support for concurrent access to data introduces a delicate issue: what consistency semantics to offer and how to implement it efficiently. A strong consistency semantics makes reasoning about concurrency easier and simplifies application development, however it is difficult to implement efficiently in a distributed environment without sacrificing performance. A weak consistency semantics on the other hand has a much higher potential to achieve better performance levels under concurrency, however it provides less guarantees, which is insufficient for some applications. Therefore, it is important to find the right trade-off.

Fault tolerance. Faults are unavoidable at large scale, because a large number of components are present that need to interact with each other. Therefore, in order to be reliable, a storage system needs to tolerate faults. One important challenge in this context is the need to handle faults transparently: they are supposed to be detected and repaired automatically, by a self-healing mechanism such that the application needs not be aware of them happening.

Security. Security is not a major concern for distributed systems that are isolated from the outside and are supposed to be accessed by trusted users only. However, with the emergence of grid computing and cloud computing, storage systems can spread over untrusted open networks (Internet) and may need to serve untrusted users. In this context, security becomes a critical issue: not only is it important to verify that users are indeed who they claim to be (authentication), but it also necessary to enforce permissions and policies that define and limit the way users can access data.

In the rest of this chapter, we present several existing approaches to data storage in distributed systems, insisting on the issues mentioned above. We conclude with a series of

limitations of these approaches and new challenges that arise in this context.

3.1 Centralized file servers

The most basic form of data storage is *centralized* data storage. In this setting, all data accesses that are issued in the distributed system are handled by a single dedicated machine that specializes to store the data and serve the access requests to it.

This dedicated machine is typically a network file server or database server, that has direct access to several block-based storage devices, such as hard-drives or solid state devices. The server is responsible to manage the storage space of the devices and to expose a file-based or higher level I/O access API to the clients. This approach is commonly referred to as *network-attached storage (NAS)*.

Direct access to the block devices is provided by several technologies. The simplest of them is *direct-attached storage (DAS)* [87], which interconnects the block devices with the server directly through the I/O bus, via SCSI or ATA/IDE. Such an approach has the advantage of enabling high performance for a low price, however, there is a significant drawback: only a very limited number of devices can be accessed simultaneously, most often not more than 16. To address the connectivity limits of DAS, *storage area networks (SANs)* [32, 166] were introduced, which feature a high performance switching hardware that enables both fast access to, as well as scalable interconnect of a large number of storage devices. A SAN however is more expensive to buy and more difficult to maintain than a DAS.

In order to expose a higher level API to the clients, a NAS typically uses standardized protocols, such as the Network File System protocol (NFS) [150], which allows the clients to access the data in the same way as local file systems are accessed. Like many other networking protocols, NFS is built on top Open Network Computing Remote Procedure Call (ONC RPC), a standardized remote procedure call convention and is described in detail in RCF 3530 [143].

Among the main advantages of centralized data storage are *simplicity* and *low cost*. Indeed, setting up a file server in the cluster for data storage is a straightforward process that requires little effort and greatly simplifies the design of the storage architecture. Since all I/O traffic is handled by a single machine, transparency, consistency semantics and security are not a concern. These advantages, combined with the low acquisition and maintenance cost, make a centralized solution desirable for small clusters where it can satisfy data storage requirements.

However, centralized data storage has important drawbacks: it features a poor scalability and a poor fault-tolerance. The dedicated server can quickly become a bottleneck when a large number of clients simultaneously try to access the data. At the same time, it is a single point of failure in the distributed system that makes access to data completely unavailable in case the dedicated server goes down.

Nevertheless, centralized solutions are extremely popular even in large-scale distributed computing projects that are predominantly compute-intensive and manipulate modest amounts of data, such as SETI@home [4].

3.2 Parallel file systems

Parallel file systems aims at addressing the poor scalability and poor fault-tolerance of centralized approaches, while retaining transparency.

In order to do so, a parallel file system employs multiple servers that are each responsible to manage a set of individual storage resources The clients do not have direct access to the underlying storage resources, but interact over the network with the servers using a standardized protocol.

The main idea behind this choice is the fact that under concurrency, the I/O workload of the clients is distributed among the servers, which greatly increases scalability, as each server has to deal with a much smaller number of clients. Furthermore, this approach makes it possible to replicate data on multiple servers, which greatly enhances fault-tolerance, as data is available in alternate locations if a server goes down.

Parallel file systems typically aim at at compatibility with the *POSIX* [42] file access interface. This choice has a major advantage: POSIX is highly standardized and therefore it enables a high degree of transparency, allowing applications to use the parallel file system as if it were a local file system. However, the choice of using POSIX as the access model also introduces important limitations: POSIX is locking-based and as such it can lead to poor performance under specific concurrent access patterns, such as reading while writing in overlapping regions of the same file.

Lustre. A massively parallel file system, *Lustre* [40] is generally used for large-scale cluster computing. An open-source project, it can aggregate Petabytes of storage capacity and can provide high levels of performance even in the range of tens of thousands of nodes. Lustre exposes a standard POSIX access interface to the clients that supports locking-based concurrent read and write operations to the same file. It employs a centralized metadata management scheme through a *metadata target (MDT)*, which is a server responsible to manage file names, directories, permissions, file layout, etc. The contents of the files is spread across *object storage servers (OSSes)* that store file data on one or more *object storage targets (OSTs)*, which are typically high-capacity disks that are accessed by the OSSes through a SAN. Thus, the aggregated capacity of a Lustre deployment is the sum of the capacities of the OSTs. For security and fault-tolerance reasons, clients are not allowed to access the OSTs directly and must do so through the OSSes.

PVFS. Designed as a high performance cluster file system for parallel applications, *PVFS* [27] specifically targets scenarios where concurrent, large I/O and many file accesses are common. To this end, PVFS distributes both data and metadata over a fixed set of storage servers, avoiding single points of contention and enabling scalability to a large number of clients. In order to ensure scalability, PVFS avoids complex locking schemes present in other parallel file systems by ordering operations in such way that they create a sequence of states that represent consistent file system directory hierarchies. For example, to create a file, data is written first on the servers, followed by metadata, and finally the corresponding entry is created in the directory. If any step fails during the file creation, no change to the file system happens, as the file is registered only in the last step. Already written data and metadata is not harmful and can be discarded at a later point. While this simple scheme has a much

higher potential to scale than a locking scheme, it comes at a cost: write/write concurrent accesses to overlapping regions of the same file are not supported. Although not explicitly forbidden, the effects of attempting to do so are undefined.

GPFS. The *General Parallel File System (GPFS)* [140], developed by IBM is a closed-source, high-performance file system that is in use by many supercomputers around the world. Files written to GPFS are split into blocks of less than 1 MB, which are distributed across multiple file servers that have direct access to several disk arrays. To prevent data loss, such blocks are either replicated on the same server using native RAID or on different servers. Metadata describes the file layout in terms of blocks, and the directory structure is distributed as well and efficiently supports a large number of files in the same directory. The clients can access files through an access interface that implements full POSIX semantics, including locking for exclusive file access thanks to a distributed locking scheme. An interesting feature of GPFS is its ability to be partition aware. More specifically, network failures that cut communication between file servers and partition them into groups, are detected through a heartbeat protocol and measures are taken to reorganize the file system such that it comprises the largest group, effectively enabling a graceful degradation.

Ceph. With the evolution of storage technologies, file system designers have looked into new architectures that can achieve scalability. The emerging *object storage devices (OSDs)* [95] couple processors and memory with disks to build storage devices that perform low-level file system management (such as block allocation and I/O scheduling) directly at hardware level. Such "intelligent" devices are leveraged by *Ceph* [167], a cluster file system specifically designed for dynamic environments that exhibit a wide range of workloads. Ceph decentralizes both data and metadata management, by using a flexible distribution function that places data objects in a large cluster of OSDs. This function features uniform distribution of data, consistent replication of objects, protection from device failures and efficient data migration. Clients can mount and access a Ceph file system through a POSIX-compliant interface that is provided by a client-side library.

3.3 Data grids

With the introduction of grid computing, presented in Section 2.2, the need arised to manage large data collections that are distributed worldwide over geographically distant locations. To address this need, data grids [165] emerged as the platform that combines several wide-area management techniques with the purpose of enabling efficient access to the data for the participants of the grid.

3.3.1 Architecture

Data grids are organized in a layered architecture, as proposed in [49, 9]. Each layer builds on the lower level layers and interacts with the components of the same level to build a complete data management system. We briefly introduce these layers, from the lowest to the highest:

Data fabric: consists of the resources that are owned by the grid participants and are involved in data generation and storage, both with respect to the hardware (file servers, storage area networks, storage clusters, instruments like telescopes and sensors, etc.) as well as the software that leverages them (distributed file systems, operating systems, relational database management systems, etc.)

Communication: defines and implements the protocols that are involved in data transfers among the grid resources of the fabric layer. These protocols are build on several well-known communication protocols, such as TCP/IP, authentication mechanisms, such as Kerberos [72], and secure communication channels, such as Secure Sockets Layer (SSL).

Data Grid Services: provides the end services for user applications to transfer, manage and process data in the grid. More specifically, this layer is responsible to expose global mechanisms for data discovery, replication management, end-to-end data transfers, user access right management in virtual organizations, etc. Its purpose is to hide the complexity of managing storage resources behind a simple, yet powerful API.

Applications. At this layer are user applications that leverage the computational power of the grid to process the data stored in the data grid. Several standardized tools, such as visualization applications, aim at presenting the end user with familiar building blocks that speed up application development.

3.3.2 Services

The need to manage storage resources that are dispersed over large distances led to several important design choices. Two important classes of services stand out.

3.3.2.1 Data transport services.

A class of services, called *data transport services* was designed that departs from data access transparency, enabling applications to explicitly manage data location and transfers, in the hope that application-specific optimizations can be exploited at higher level. The focus of such services is to provide high performance end-to-end transfers using low overhead protocols, but this approach places the burden of ensuring data consistency and scalability on the application.

Data transport is concerned not only with defining a communication protocol that enables two end-to-end hosts to communicate among each other with the purpose of transferring data, but also with other higher level aspects such as the mechanisms to route data in the network or to perform caching in order to satisfy particular constraints or speed up future data access. Several representative services are worth mentioning in this context.

Internet Backplane Protocol (IBP). *IBP* [15] enables applications to optimize data transfers by controlling data transfers explicitly. Each of the nodes that is part of the IBP instance has a fixed-size cache into which data can be stored for a fixed amount of time. When data is routed during an end-to-end data transfer, data is cached at intermediate locations in a manner similar to “store-and-forward”. The application has direct control over the caches

of IBP nodes and can specify what data to cache where, which increases the chance of future requests for the same data to find it in a location that is close to where the data is required. IBP treats data as fixed-size byte arrays, in a similar fashion as the Internet Protocol, which splits data into fixed-size packets. The same way as IP, it provides a global naming scheme that enables any IBP node to be uniquely identified. Using this global naming scheme, applications can move data around without caring about the underlying storage of individual nodes, which is transparently managed by IBP.

GridFTP. *GridFTP* [2, 20] extends the default FTP protocol with features that target efficient and fast data transfer in grid environments, where typically large files need to be transferred between end points. Like FTP, GridFTP separates effective data transfers from control messages by using a different communication channel for each of them. This enables third-party file transfers that are initiated and controlled by an entity that is neither the source, nor the destination of the transfer. In order to support large files better, GridFTP provides the the ability to stripe data into chunks that are distributed among the storage resources of the grid. Such chunks can be transferred in parallel to improve bandwidth utilization and speed up transfers. GridFTP can also use multiple TCP sockets over the same channel between a source and a destination in order to improve bandwidth utilization further in wide-area settings.

3.3.2.2 Transparent data sharing services.

Since grid participants share resources that are dispersed over large geographical areas, data storage needs to adapt accordingly. Unlike data transport services where data access is managed explicitly at application level, several attempts try to provide transparent access to data, in a manner similar to parallel file systems, but at global grid scale. This approach has the advantage of freeing the application from managing data locations explicitly, but faces several challenges because resources are heterogeneous and distances between them can vary greatly.

Replication becomes crucial in this context, as it improves locality of data and preserves bandwidth, greatly increasing scalability and access performance. However, on the down side, consistency among replicas that are stored in geographically distant locations becomes a difficult issue that is often solved by choosing a weak consistency model.

Grid Data Farm (Gfarm). Gfarm [155] is a framework that integrates storage resources and I/O bandwidth with computational resources to enables scalable processing of large data sizes. At the core of Gfarm is the Gfarm file system, which federates local file systems of grid participants to build a unified file addressing space that is POSIX compatible and improves aggregated I/O throughput in large scale settings. Files in Gfarm are split into fragments that can be arbitrarily large and can be stored in any storage node of the grid. Applications may fine-tune the number of replicas and replica locations for each file individually, which has the potential to avoid bottlenecks to frequently accessed files and to improve access locality. Furthermore, the location of fragment replicas is exposed through a special API at application level, which enables to schedule computations close to the data.

JuxMem. Inspired by both both DSM systems and P2P systems, *JuxMem* [7] is a hybrid data sharing service that aims to provide location transparency as well as data persistence in highly dynamic and heterogeneous grid environments. Data is considered to be mutable (i.e., it is not only read, but also concurrently updated) and is replicated in order to improve data access locality and fault tolerance. In this context, ensuring replica consistency is a difficult issue. JuxMem proposes an approach based on group communication abstractions to ensure entry-consistency, while guaranteeing high data availability and resilience to both node crashes and communication failures.

XtreemFS. *XtreemFS* [64] is an open-source distributed file system that is optimized for wide-area deployments and enables clients to mount and access files through the Internet from anywhere, even by using public insecure networking infrastructure. To this end, it relies on highly secure communication channels built on top of SSL and X.509. XtreemFS exposes a configurable replication management system that enables easy replication of files across data centers to reduce network consumption, latency and increase data availability. Several features aim at dealing with high-latency links that are present in wide-area networks: metadata caching, read-only replication based on fail-over replica maps, automatic on-close replication, POSIX advisory locks.

3.4 Specialized storage services

With data sizes growing and distributed applications gaining in complexity, the traditional POSIX file system access interface becomes a limitation for data management. The main disadvantage of POSIX is the fact that it is designed as an all-purpose access interface that is not aware of the specific application access patterns, which greatly limits the potential to introduce optimizations in this direction and improve scalability. For this purpose, several specialized file systems and storage services have been introduced that depart from POSIX.

3.4.1 Revision control systems.

Revision control systems specialize in the automated management of changes to collections of documents, programs, and other information stored as computer files. This is highly relevant for collaborative development, where large groups of individuals share and concurrently update the same files. In this context, the most important problem that needs to be solved is *revision*: how to offer a flexible and efficient mechanism to apply changes to a file, such that it is easy to revoke them later if necessary. To solve this problem, revision control systems keep an annotated history of changes that enables them to reconstruct any past state of the files under its control. The history of changes must not necessarily be linear, enabling a file to evolve in many directions (referred to as *branching*) that can be eventually merged together in a single direction. Users explicitly control the submission of changes, branching and merging. In order to remain scalable under these complex circumstances, revision control systems usually avoid synchronization mechanisms and enable users to perform their changes in isolation. Potential consistency issues are detected only when the changes are submitted and, if present, they need to be solved by the users manually. Examples of revision control system are listed below.

SVN. Subversion (SVN) [171] is a popular revision control system that maintains full versioning for directories, renames, file metadata, etc. Users are allowed to change, move, branch and merge entire directory-trees very quickly, while retaining full revision history. SVN is based on a client-server model, where users synchronize in though a centralized server that is responsible to maintain changeset and revision history. This choice has an important advantage: user have a unified view over the whole codebase, which simplifies several management tasks (such as testing) easy. However, the centralized approach has limited scalability potential and represents a single point of failure.

Git. Unlike SVN, *Git* [83] is a distributed revision control system that is based on a peer-to-peer approach to store the codebase. Rather than a single, central server through which all clients synchronize, in Git each peer holds its own local copy of the repository. Revision is conducted by exchanging patches between peers using gossip protocols, which means that there is no global reference copy of the codebase, only working copies that eventually become consistent. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around between various peers. In order to support this pattern efficiently, Git supports rapid branching and merging, and includes dedicated tools to visualize and browse dependencies. The distributed nature of Git makes it great for large projects where users typically need to access and change only small parts for which they are responsible. However, tasks such as obtaining the most recent view of the codebase are rather difficult due to the slow propagation of changes.

3.4.2 Versioning file systems.

Unlike revision control systems that are designed to enable users to control versioning explicitly, versioning is handled by versioning file systems in a *transparent* fashion. More precisely, from the user point of view, the file system behaves like a regular file system. However, at regular time intervals or when other conditions apply, the file system consolidates all recent changes into a new snapshot that can be later accessed for reference. Unlike revision control systems, versioning file systems support a linear evolution only. Versioning file systems are typically used for archival purposes, when reference data needs to be kept for predefined amounts of time. In this context, versioning transparency is a great advantage, as it enables applications to run unmodified, while keeping track of their history of accesses to data.

Fossil. Built as an archival file server, *Fossil* [122] maintains a traditional file system on the local disk and periodically archives snapshots of this file system to *Venti* [123], a write-once, read-many archival block storage repository that is installed in conjunction with it. Snapshots in Fossil are placed in special directories and can be directly accessed through the standard file system interface that obeys POSIX semantics. The active file system is presented as "/active", while past snapshots are presented as "/snapshot/date/time". To implement snapshots, Fossil keeps track of the time when each file system block was changed. If a write occurs on a block that was not changed in predefined amount of time, then the block is treated as immutable and a new block is generated in a copy-on-write fashion. Otherwise, the block is simply overwritten as in a regular file system. When the predefined amount

of time expired, all modified blocks are transferred to Venti and consolidated into a new snapshot of the file system.

ZFS. The *Zettabyte File System* [58] is a versioning file system developed by Sun Microsystems that is designed to efficiently leverage large storage capacities. ZFS uses a copy-on-write transactional model in which blocks are never overwritten but instead a new copy of the block is created in an alternate location and then the modified data is written to it. Any metadata blocks that reference such newly written blocks are recursively treated in the same fashion until a new snapshot of the whole file system is obtained that shares unmodified data with the previous snapshots. ZFS snapshots are created very quickly, since all the data composing the snapshot is already stored. They are also space efficient, since any unchanged data is shared among the file system and its snapshots. ZFS introduces several interesting features, such as writable snapshots and dynamic striping of data. However, an important disadvantage of ZFS is the fact that it is not designed to enable concurrent access to data in a distributed environment, and, as such, is limited to be used by a single client at a time in the same way as a local file system.

3.4.3 Dedicated file systems for data-intensive computing

With the emergence of data-intensive computing, several paradigms, such as *MapReduce* [38], appeared that exploit the parallelism at data level in order to scale even for huge data sizes. In this context, data storage is faced with specific access patterns: highly concurrent reads from the same file at fine-grain level, few overwrites, highly concurrent appends to the same file. These access patterns prompted the development of several distributed file systems:

GoogleFS. The *Google File System (GoogleFS)* [53] is a proprietary distributed file system in development by Google as a response to its data storage needs. GoogleFS leverages large clusters of commodity hardware to provide high-throughput access to data. Files in GoogleFS are traditionally very large, reaching hundreds of GB. Each file is is split into 64 MB chunks, which are replicated and distributed among the nodes of the cluster. A centralized metadata server, called the *Master*, is responsible to manage the directory hierarchy and the layout of each file (what chunks make up the file and where they are stored). A specialized API is provided that enables clients to read from and write/append to the same file in a highly concurrent fashion. Append operations are atomic: they guarantee that the contents of the appended data will appear somewhere in the file as a contiguous sequence of bytes. However, the precise location is not known in advance. Concurrent write operations are also supported, but, as is the case with PVFS, the final result of doing so is undefined. In order to enable concurrent write and append operations, the Master employs a system of time-limited, expiring "leases", which guarantee exclusive permission to a process to modify a chunk. The modifications are always processed by the server that holds the primary copy of the chunk. It is the responsibility of this server to propagate the modifications to the other servers that hold the replicas of the chunk. In order to ensure consistency among replicas, a leader election protocol is employed: no modification is applied unless all servers that hold a replica of the chunk acknowledge the modification.

HDFS. The *Hadoop Distributed File System (HDFS)* [144] is the primary storage system used by *Hadoop* [169], a popular open-source *MapReduce* [38] framework that has seen a wide adoption in the industry, with Yahoo! and Facebook counting among its users. HDFS was modeled after GoogleFS: files are split into 64 MB chunks that are distributed among *datanodes*. A centralized server, called the *namenode*, is responsible for metadata management. Access to files is facilitated by a specialized Java API that enables concurrent reads and appends to the same file, in the same sense as GoogleFS. However, unlike GoogleFS, data is considered immutable: once written, it cannot be modified. Moreover, while the API offers support for concurrent appends, in practice this feature is not yet implemented. The namenode is a single point of failure for an HDFS installation. When it goes down, the whole file system is offline. When it comes back up, the name node must replay all outstanding operations, which can take a long time on large clusters.

3.4.4 Cloud storage services

Cloud storage is a storage model that emerged in the context of cloud computing. Cloud providers operate large data centers, whose storage resources provide huge aggregated capacities. These resources are virtualized according to the requirements of the customer and exposed through a storage service. Customers can upload and download files and data objects to and from the cloud through this storage service, paying only for the occupied storage space. Data availability and fault tolerance are key issues in the design of cloud storage services, as the provider needs to be able to offer strong guarantees to the customers through the service level agreement in order to make cloud storage an attractive offer. Several such cloud storage services stand out:

Amazon Dynamo. *Dynamo* [39] is a highly available key-value store internally developed at Amazon that is used to power Amazon's core services and to provide an "always-on" experience. In order to achieve a high level of availability, Dynamo replicates all key-value pairs to multiple hosts, whose number can be configured independently for each pair. Each replica is assigned to a coordinator node, which is responsible to maintain its replication factor constant. A consistent hashing scheme is used for replica placement. Replication itself is performed in a fully asynchronous fashion, at the expense of sacrificing consistency under certain fault scenarios. More specifically, under no faults, updates will be eventually propagated to all replicas. However, faults (such server outages or partitions) can delay the propagation of updates for extended periods of time. This might lead to inconsistencies which are handled by Dynamo through object versioning: each update creates a new and immutable version of the object. Most of the time, the system can automatically determine how to merge versions of the same object into a single, consistent object. However, under faults combined with heavy concurrency, objects may evolve in different directions and a semantic reconciliation at application level is necessary, for which Dynamo exposes a specific API.

Amazon S3. *Amazon S3* [130] is a web service that partially builds on *Dynamo* [39] to enable high-availability cloud storage to Amazon clients. S3 is intentionally designed with a minimal feature set that however enables reaching high availability rates, which are guaranteed by Amazon through the service level agreement. The access interface enables clients

to write, read and delete objects whose size ranges from 1 byte to 5 GB. Unlike file systems, objects are not named and placed in a hierarchic directory structure, but are rather assigned an unique *ID* and placed in a bucket, which together with the *ID* can be used to retrieve the object later. Partial writes are not supported: each write request fully replaces the object content with the written data. Under concurrency, there is no way to determine the writer that succeeds in replacing the object content the last. However, partial reads of the same object are supported.

Azure BLOB Storage Service. The *Azure BLOB storage service* [126] is a storage service specifically designed for the Microsoft Azure cloud platform. It provides support to store and retrieve binary large objects (BLOBs) through the REST API [44]. Each BLOB must belong to a container, which is created through the REST API as well. Two types of BLOBs are supported: *block BLOBs*, which are optimized for streaming, and *page BLOBs*, which are optimized for random read/write operations on ranges of bytes. A block BLOB can either be created in a single step if it smaller that 64 MB, or in multiple steps by transferring chunks of at most 4 MB at a time to the cloud. In this last case, an extra API call is used to consolidate the chunks into a single block BLOB, which can reach at most 200 GB. Page BLOBs have fixed sizes, which are specified at creation time, and are initially zero-filled. The client can then write the desired content in the BLOB using the write primitive. Page BLOBs are limited to 1 TB. An interesting feature of the Azure BLOB storage service is the ability to perform conditional updates, i.e. execute a write only if a particular condition, specified in the write operation, holds.

3.5 Limitations of existing approaches and new challenges

We summarize the approaches presented so far in Table 3.1. For each type of storage systems, we have selected a few representative examples and classified them according to a series of criteria that have the high impact on performance and scalability. More precisely, we synthesize two aspects: (i) whether data and metadata is organized in a centralized or decentralized fashion, which hints at potential bottlenecks that limit scalability; (ii) how concurrency control is implemented and what guarantees are offered. Also included are details about versioning: whether it is supported and if so, how it is handled.

With growing data management requirements, existing data storage systems face many limitations and need to address new challenges that arise in this context. In the rest of this section, we briefly discuss some of these limitations.

Too many files. Most distributed applications process data objects that are typically small, often in the order of KB: text documents, pictures, web pages, scientific datasets, etc. Each such object is typically stored by the application as a file in a distributed file system or as an object in a specialized storage service. A similar scenario occurs for tightly-coupled applications that consist of billions of threads that perform parallel computations. Usually, there is a need to save intermediate results at regular intervals, which, out of the need to avoid I/O synchronization, is done is done independently for each thread in a separate file.

However, with the number of files easily reaching the order of billions [53], a heavy burden on the underlying storage service, which must efficiently organize the directory struc-

Architecture	Example	Data / Meta-data	Concurrency control	Versioning
Centralized	NFS server	c/c	lock daemon	not supported
Parallel file systems	Lustre	d/c	hybrid locking using leases	not supported
	GPFS	d/d	distributed locking scheme	not supported
	PVFS	d/d	none: undefined overlapped W/W	not supported
Grid data sharing services	Gfarm	d/c	none: undefined overlapped W/W	not supported
	JuxMem	d/d	acquire-release; entry consistency	not supported
	XtreemFS	d/c	POSIX advisory locks	not supported
Revision control	SVN	c/c	manual conflict resolution; branch & merge	changeset
	Git	d/d	manual conflict resolution; branch & merge	changeset
Versioning file systems	Fossil	c/c	single client at a time	snapshot
	ZFS	d/c	single client at a time	snapshot
Data-intensive	GoogleFS	d/c	locking using leases; atomic concurrent appends	not supported
	HDFS	d/c	single writer; immutable data	not supported
Cloud storage	Amazon S3	d/d	atomic replace of full object	snapshot
	Dynamo	d/d	master replica; auto-branching under faults; semantic reconciliation	snapshot
	Azure	d/d	conditional updates	not supported

Table 3.1: Summary of approaches presented so far. In the third column, *c* stands for centralized and *d* stands for decentralized

ture such that it can lookup files in a timely manner. Therefore, one important challenge in this context is to find scalable ways of organizing data such that it does not lead to complex namespaces that are slow to browse and maintain.

Centralized metadata management. Metadata is used by storage systems to maintain critical information about the data itself: directory hierarchy, file names and attributes, permissions, file layout (what parts make up the file and where they are stored), etc. It is an essential means to ensure data access transparency. Since metadata is a tiny fraction of the size of the data itself, the details of metadata management might not seem critical. However, at very large scale even a tiny fraction of the total data that is stored can quickly reach huge sizes.

A study conducted by Yahoo [145] for example, shows that their data requirements are quickly approaching the Zettabyte limit. Most of their data is stored on HDFS, which therefore needs to scale to the Zettabyte order, both in terms of storage space and access performance. HDFS however is built upon a single-node namespace server architecture, which acts as a single container of the file system metadata. In order to make metadata operations fast, the name-node loads the whole metadata into its memory. Therefore, the total size of the metadata is limited by the amount of RAM available to the name-node. Since it was established that the overhead of metadata represented in RAM is 1 GB for each 1 PB of actual data, it is easily observable that the current design does not scale to meet the Zettabyte requirement.

Many storage systems presented in this chapter are in the same situation as HDFS, relying on centralized metadata management: Lustre, GoogleFS, etc. An important challenge that arises in this context is thus to find scalable ways of managing the metadata. Decentralization seems to promise a lot of potential in this direction, but introduces metadata consistency issues.

Limited throughput under heavy access concurrency. As distributed systems increase in scale and complexity, concurrency control plays a crucial role. Several systems presented so far aim at POSIX compatibility, as it is a widely accepted standard that offers high compatibility with a broad range of applications. Over-generalization however has its price: such systems need to implement complex locking schemes in order to satisfy POSIX consistency semantics, which under many circumstances is a limitation of scalability, as it leads to aggregated throughputs that are well below expected numbers. For example, this is the reason why many large scale applications that consist of a large number of threads prefer to write many small independent files rather that write concurrently in the same file.

To deal with this issue, many approaches over-relax consistency semantics even to the point where it remains undefined: for example, in PVFS the outcome of concurrent overlapping writers in the same file is not known. While this approach certainly simplifies concurrency control overhead and enables reaching high throughputs, in many applications such undefined behavior is insufficient.

Therefore, an important challenge in this context, is to define a consistency semantics and implement a concurrency control mechanism for it that enables reaching high throughputs under concurrency, while offering a clear set of guarantees.

Limited versioning capabilities under concurrency. Many existing versioning file systems are rich in features, but are not designed to be used at large scale under heavy access concurrency. For example, ZFS and Fossil are local file systems that can be mounted and used only by single client at a time. In a distributed environment, this limitation is clearly a bottleneck. Revision control services also suffer from the same problem. For example, SVN uses a centralized server to store the repository, which is at large scale again a bottleneck. While Git tried to address this issue by proposing a decentralized P2P model, updates propagate slow at global scale, which limits feasibility to a restricted set of usage patterns.

On the other hand, cloud storage services are designed to scale and support concurrent, distributed access to data. However, they have only limited support for versioning. For example, Amazon S3 supports versioning for its objects, but objects are limited to a size of 5GB. Moreover, there is no support to apply partial updates to an object: an object can only be fully replaced by a newer version in an atomic fashion. This limits the potential of versioning, because an object cannot hold the combined effect of many concurrent writers.

Thus, another important challenge that needs to be addressed is how to provide efficient versioning support under concurrency.

Conclusions. Extensive work has been done in the area of data storage in large-scale, distributed systems. We presented several approaches and underlined a series of limitations and new challenges that these approaches face under growing pressure of increasing data storage demands. The rest of this thesis aims at addressing several of these new challenges, both by combining ideas from existing approaches as well as proposing novel data management techniques that are designed from scratch with these challenges in mind.

Part II

BlobSeer: a versioning-based data storage service

Chapter 4

Design principles

Contents

In the previous chapter, we introduced several important issues that need to be considered when designing a storage system. We presented several existing approaches and underlined their limitations and the new challenges that arise in the context of data storage. This chapter focuses on a series of core design principles that we propose in order to overcome many of these limitations and address several challenges. We insist in particular on the importance of *versioning* as a crucial feature that enhances data access concurrency, ultimately leading to better scalability and higher performance.

4.1 Core principles

4.1.1 Organize data as BLOBs

We are considering applications that process huge amounts of data that are distributed at very large scales. To facilitate data management in such conditions, a suitable approach is to

organize data as a set of *huge objects*. Such objects (called BLOBs hereafter, for *Binary Large OBjects*), consist of long sequences of bytes representing unstructured data and may serve as a basis for transparent data sharing at large-scale. A BLOB can typically reach sizes of up to 1 TB. Using a BLOB to represent data has two main advantages.

Scalability. Applications that deal with fast growing datasets that easily reach the order of TB and beyond can scale better, because maintaining a small set of huge BLOBs comprising billions of small, KB-sized application-level objects is much more feasible than managing billions of small KB-sized files directly [53]. Even if there was a distributed file system that would support access to such small files transparently and efficiently, the simple mapping of application-level objects to file names can incur a prohibitively high overhead compared to the solution where the objects are stored in the same BLOB and only their offsets need to be maintained.

Transparency. A data-management system relying on globally shared BLOBs uniquely identified in the system through global *IDs* facilitates easier application development by freeing the developer from the burden of managing data locations and data transfers explicitly in the application.

However, these advantages would be of little use unless the system provides support for *efficient fine-grain access* to the BLOBs. Large-scale distributed applications typically aim at exploiting the inherent data-level parallelism. As such, they usually consist of concurrent processes that access and process small parts of the same BLOB in parallel. At the minimal level, this means it must be possible to create a BLOB, read/write subsequences of bytes from/to the BLOB at arbitrary offsets and append a sequence of bytes to the end of the BLOB. The storage service needs to provide an efficient and scalable support for a large number of concurrent processes that access such subsequences, whose size can go in some cases as low as in the order of KB. If exploited efficiently, this feature introduces a very high degree of parallelism in the application.

4.1.2 Data striping

Data striping is a well-known technique to increase the performance of data accesses. Each BLOB is split into *chunks* that are distributed all over the nodes that provide storage space. Thereby, the I/O workload is distributed among multiple machines, which enables the system to scale and reach a high performance level. Two important features need to be taken into consideration in order to maximize the benefits of accessing data in a distributed fashion.

Configurable chunk distribution strategy. The distribution of chunks in the system has a high impact on the benefits that an application can reap from data-striping. For example, if a client needs to read a subset of chunks that was previously written to the same provider, data striping is ineffective because the chunk distribution not optimized for the way the application reads back data. In order to deal with this issue, the chunk distribution strategy needs to be adapted to the application needs. Most of the time, load-balancing is highly desired, because it enables a high aggregated throughput when different parts of the BLOB are simultaneously accessed. However, a lot of

more complex scenarios are possible. For example, green computing might introduce an additional objective like minimizing energy consumption by reducing the number of storage space providers. Given this context, an important feature is to enable the application to configure the chunk distribution strategy such that it can optimally leverage data-striping according to its own access patterns.

Dynamically adjustable chunk sizes. The performance of distributed data processing is highly dependent on the way the computation is partitioned and scheduled in the system. There is a trade-off between splitting the computation into smaller work units in order to parallelize data processing, and paying for the cost of scheduling, initializing and synchronizing these work units in a distributed fashion. Since most of these work units consist in accessing data chunks, adapting the chunk size is crucial. If the chunk size is too small, then work units need to fetch data from many locations, potentially canceling the benefits of techniques such as scheduling the computation close to the data. On the other hand, selecting a chunk size that is too large may force multiple work units to access the same data chunks simultaneously, which limits the benefits of data distribution. Therefore, it is highly desirable to enable the application to fine-tune how data is split into chunks and distributed at large scale.

4.1.3 Distributed metadata management

Since each massive BLOB is striped over a large number of storage space providers, additional metadata is needed in order to remember the location of each chunk in the BLOB, such that it is possible to map subsequences of the BLOB defined by *offset* and *size* to the corresponding chunks. This need results from the fact that without additional metadata, the information about the internal structure of the BLOB is lost. Since BLOBs can reach huge sizes and need to support fine-grain data-striping, metadata becomes an important issue.

As mentioned in Section 3.5, many distributed file systems such as GoogleFS [53] and HDFS [144], which are used in large-scale production environments, are on the verge of reaching their limits because they use a centralized metadata management scheme. Thus, we argue for a distributed metadata management scheme (presented in detail in Chapter 6), which brings several advantages to such an approach.

Scalability. A distributed metadata management scheme potentially scales better that a centralized approach, both with respect to increasing metadata sizes and concurrent access to metadata. This is a consequence of the fact that the I/O workload associated to metadata overhead can be distributed among the metadata providers, which means a higher aggregated metadata storage space and a lower pressure on each metadata provider. Both properties are important in order to efficiently support fine-grain access to the BLOBs.

Data availability. A distributed metadata management scheme enhances data availability, as metadata can be replicated and distributed to multiple metadata providers. This avoids letting a centralized metadata server act as a single point of failure: the failure of a particular node storing metadata does not make the whole data unavailable.

4.1.4 Versioning

We defend the use of versioning as a core principle for data management. Versioning certainly implies that multiple copies of the same data needs to be kept. This leads to significant storage space overheads when compared to traditional approaches that keep only the latest version. However, storage space is becoming increasingly cheaper and data-centers are designed in a scalable fashion that enables them to grow easily. Therefore, versioning has the potential to bring high benefits for a low price.

Enhanced data access parallelism. One of the main advantages of versioning is the potential to enhance parallel access to data, both when versions are manipulated transparently by the storage service or explicitly by the application. Unlike traditional approaches that keep only the latest version, multiple versions enable a better isolation which greatly reduces the need to synchronize concurrent access. To leverage this at a maximal extent, we propose to generate a new snapshot version of the BLOB each time it is written or appended by a client. By using the right means to generate a new snapshot, two important advantages can be obtained.

Overlapped data acquisition with data processing through explicit versioning. In many cases data is processed in two phases: a data acquisition phase where data is gathered, and a data processing phase where a computation over the data is performed. Versioning enables overlapping of these two phases: data acquisition can run and lead to the generation of new BLOB snapshots, while data processing may run concurrently on previously generated snapshots *without any synchronization*. Explicitly exposing a versioning-based data access interface to the user enables it to control the process at a fine level, e.g., by specifying which snapshot generated by the acquisition phase should be processed.

High-throughput data access under concurrency through immutable data /metadata. Versioning enables better performance levels under concurrency evens when the applications do not explicitly leverage multiple versions. For example, the storage service can rely on versioning internally in order to optimize concurrent access to the same BLOB. In this context, we propose to *keep data and metadata immutable*. Doing so enables the storage service to break reads and writes from/to the BLOB into elementary operations that are highly decoupled and require a minimal amount of synchronization. For example, a read operation that needs to access a chunk does not need to wait for a concurrent write operation that updates the same chunk, because the write operation does not overwrite the chunk.

Archival storage. In today's era of exponential growth of information, storing old data is important for reference purposes. However, with growing processing power it becomes also an important source for data mining applications that can track its evolution in time and infer new knowledge. In this context, versioning is a powerful tool that enables efficient management of archival data. Moreover, as mentioned in Chapter 3, in many cases legislation requires keeping an auditable trail of changes made to electronic records, which is a complex issue to manage at application level without versioning support. Generating a

new snapshot of the BLOB with each write and append facilitates tracking such changes at the smallest possible granularity, greatly enhancing security and compliance with the legislation.

While versioning certainly has important benefits, it is not easy to efficiently implement it in practice. We identified several important factors that need to be taken into consideration.

Differential updates. The generation of a new BLOB snapshot version for each write or append might seem to lead to a huge waste of storage space, especially when BLOBs are large and only small parts of them are updated. We propose the use of differential updates: the system can be designed is such way that it only stores the difference with respect to the previous versions. This eliminates unnecessary duplication of data and metadata, and saves storage space. The new snapshot shares all unmodified *data* and *metadata* with the previous versions, while creating the illusion of an independent, self-contained snapshot from the client's point of view.

Atomic snapshot generation. A key property when providing versioning support at application level is *atomicity*. Readers should not be able to access transiently inconsistent snapshots that are in the process of being generated. This greatly simplifies application development, as it reduces the need for complex synchronization schemes at application level.

Asynchronous operations. I/O operations in data-intensive applications are frequent and involve huge amounts of data, thus latency becomes an issue. Over the last years, asynchronous, loosely-coupled system designs [30] have gained popularity in distributed computing, because they tolerate high latencies better than synchronous systems, and enable scalability up to a larger number of participants. An asynchronous access interface can help the application to hide data access latency by offering support to overlap the computation with the I/O.

4.2 Versioning as a key to support concurrency

We defended versioning as a key mechanism to support efficient access to a BLOB under heavy access concurrency. In this section, we zoom on versioning further, detailing how it can be leveraged to enhance the performance of concurrent accesses.

4.2.1 A concurrency-oriented, versioning-based access interface

We have argued in favor of a BLOB access interface that needs to be *asynchronous*, *versioning-based* and which must guarantee *atomic generation* of new snapshots each time the BLOB gets updated.

To meet these properties, we propose a series of primitives. To enable asynchrony, control is returned to the application immediately after the invocation of primitives, rather than waiting for the operations initiated by primitives to complete. When the operation completes, a callback function, supplied as parameter to the primitive, is called with the result of the operation as its parameters. It is in the callback function where the calling application takes the appropriate actions based on the result of the operation.

4.2.1.1 Basic BLOB manipulation

```
1 CREATE(callback(id))
```

This primitive creates a new empty BLOB of size 0. The BLOB will be identified by its `id`, which is guaranteed to be globally unique. The callback function receives this id as its only parameter.

```
1 WRITE(id, buffer, offset, size, callback(v))
2 APPEND(id, buffer, size, callback(v))
```

The client can update the BLOB by invoking the corresponding `WRITE` or `APPEND` primitive. The initiated operation copies `size` bytes from a local `buffer` into the BLOB identified by `id`, either at the specified `offset` (in case of write), or at the end of the BLOB (in case of append). Each time the BLOB is updated by invoking write or append, a new snapshot reflecting the changes and labeled with an incremental version number is generated. The semantics of the write or append primitive is to submit the update to the system and let the system decide when to generate the new snapshot. The actual version assigned to the new snapshot is not known by the client immediately: it becomes available only at the moment when the operation completes. This completion results in the invocation of the callback function, which is supplied by the system with the assigned version `v` as a parameter.

The following guarantees are associated to the above primitives.

Liveness: For each successful write or append operation, the corresponding snapshot is eventually generated in a finite amount of time.

Total version ordering: If the write or append primitive is successful and returns version number v, then the snapshot labeled with v reflects the successive application of all updates numbered $1 \ldots v$ on the initially empty snapshot (conventionally labeled with version number 0), in this precise order.

Atomicity: Each snapshot appears to be generated instantaneously at some point between the invocation of the write or append primitive and the moment it is available for reading.

Once a snapshot was successfully generated, its contents can be retrieved by calling the following primitive:

```
1 READ(id, v, buffer, offset, size, callback(result))
```

`READ` enables the client to read from the snapshot version `v` of BLOB `id`. This primitive results in replacing the contents of the local `buffer` with `size` bytes from `v`, starting at `offset`, if the snapshot has already been generated. The callback function receives a single parameter, `result`, a boolean value that indicates whether the read succeeded or failed. If `v` has not been generated yet, the read fails and result is false. A read fails also if the total size of the snapshot `v` is smaller than `offset` + `size`.

4.2.1.2 Learning about new snapshot versions

Note that there must be a way to learn about both the generated snapshots and their sizes, in order to be able to specify meaningful values for `v`, `offset` and `size`. This is the performed by using the following ancillary primitives:

```
1 GET_RECENT(id, callback(v, size))
2 GET_SIZE(id, v, callback(size))
```

The `GET_RECENT` primitive queries the system for a *recent* snapshot version of the blob `id`. The result of the query is the version number v which is passed to the `callback` function and the size of the associated snapshot. A positive value for v indicates success, while any negative value indicates failure. The system guarantees that: 1) $v \geq max(v_k)$, for all snapshot versions v_k that were successfully generated before the call is processed by the system; and 2) All snapshot versions with number lower or equal to v have successfully been generated as well and are available for reading. Note that this primitive is only intended to provide the caller with information about recent versions available: it does not involve strict synchronizations and does not block the concurrent creation of new snapshots.

The `GET_SIZE` primitive is used to find out the total size of the snapshot version `v` for BLOB `id`. This size is passed to the *callback* function once the operation has successfully completed.

Most of the time, the `GET_RECENT` primitive is sufficient to learn about new snapshot versions that are generated in the system. However, some scenarios require the application to react to updates as soon as possible after they happen. In order to avoid polling for new snapshot versions, two additional primitives are available to subscribe (and unsubscribe) to notifications for snapshot generation events.

```
1 SUBSCRIBE(id, callback(v, size))
2 UNSUBSCRIBE(id)
```

Invoking the `SUBSCRIBE` primitive registers the interest of a process to receive a notification each time a new snapshot of the BLOB `id` is generated. The notification is performed by calling the callback function with two parameters: the snapshot version `v` of the newly generated snapshot and its total `size`. The same guarantees are offered for the version as with the `GET_RECENT` primitive. Invoking the `UNSUBSCRIBE` primitive unregisters the client from receiving notifications about new snapshot versions for a given BLOB `id`.

4.2.1.3 Cloning and merging

`WRITE` and `APPEND` facilitate efficient concurrent updates to the same BLOB without the need to synchronize explicitly. The client simply submits the update to the system and is guaranteed that it will be applied at some point. This works as long as there are no semantic conflicts between writes such that the client needs not be aware of what other concurrent clients are writing. Many distributed applications are in this case. For example, the application spawns a set of distributed workers that concurrently process records in a large shared file, such that

each worker needs to update a different record or the worker does not care if its update to the record will be overwritten by a concurrent worker.

However, in other cases, the workers need to perform concurrent updates that might introduce semantic conflicts which need to be reconciled at higher level. In this case, `WRITE` and `APPEND` cannot be used as-is, because the state of the BLOB at the time the update is effectively applied is unknown. While many systems introduce transactional support to deal with this problem, this approach is widely acknowledged both in industry and academia to have poor availability [50]. We propose a different approach that is again based on versioning and introduce another specialized primitive for this purpose:

```
1 CLONE(id, v, callback(new_id))
```

The `CLONE` primitive is invoked to create a new BLOB identified by `new_id`, whose initial snapshot version is not empty (as is the case with `CREATE`) but rather duplicates the content of snapshot version `v` of BLOB `id`. The new BLOB looks and acts exactly like the original, however any subsequent updates to it are independent of the updates performed on the original. This enables the two BLOBs to evolve in divergent directions, much like the `fork` primitive on UNIX-like operating systems.

Using `CLONE`, workers can isolate their own updates from updates of other concurrent workers, which eliminates the need to lock and wait. At a later point, these updates can be "merged back" in the original BLOB after detecting potential semantic conflicts and reconciling them. Another primitive is introduced for this purpose:

```
1 MERGE(sid, sv, soffset, size, did, doffset, callback(dv))
```

`MERGE` takes the region delimited by `soffset` and `size` from snapshot version `sv` of BLOB `sid` and writes it starting at `doffset` into BLOB `did`. The effect is the same as if `READ(sid, sv, buffer, soffset, size, callback(result))` was issued, followed by `WRITE(did, buffer, doff, size, callback(dv))`. The only difference is the fact that `MERGE` enables the system to perform this with negligible overhead, as unnecessary data duplication and data transfers to and from the client can be avoided.

Both `CLONE` and `MERGE` can take advantage of differential updates, sharing unmodified data and metadata between snapshots of different BLOBs. Since no data transfer is involved, this effectively results in the need to perform minimal metadata updates, which enables efficient semantic-based reconciliation, as described in Section 4.2.3.

4.2.2 Optimized versioning-based concurrency control

Using the primitives presented in the previous section, versioning has the potential to enable the application to leverage concurrency efficiently. Of course, this is only useful if the storage system that implements them takes advantage of their inherent parallelism in an efficient fashion. In this section we insist on precisely this aspect, elaborating on concurrency control techniques that can be employed to do so.

Read/read and read/write. Obviously, concurrent readers never interfere with each other: they never modify any data or metadata, and cannot generate inconsistent states. Therefore, concurrent reads do not have to synchronize. Moreover, a reader can learn about new snapshots only through notifications and direct inquiry of the system. This guarantees that the snapshots the readers learn about have been successfully generated, and their corresponding data and metadata will never be modified again. Therefore, no synchronization is necessary and concurrent reads can freely proceed in the presence of writes.

Write/write. The case of concurrent writers requires closer consideration. Updating a BLOB by means of a write or append involves two steps: first, writing the data, and second, writing the metadata. We specifically chose this order for two reasons: first, to favor parallelism; second, to reduce the risk for generating inconsistent metadata in case of failures. Writing the data in a first phase and metadata in a second phase enables full parallelism for the first phase. Concurrent writers submit their respective updates to the system and let the system decide about the update ordering. As far as this first phase is concerned, concurrent writers do not interfere with each other. They can write their chunks onto data storage providers in a fully parallel fashion. Besides, if a writer fails, no inconsistency may be created, since no metadata has been created yet.

It is at metadata level where the newly written chunks are integrated in a new snapshot. It is done by generating new metadata that reference both the new chunks and the metadata of the previous snapshot versions in such way as to offer the illusion of an independent, self-contained snapshot. At a first glance, writing the metadata might not seem to be parallelizable. Indeed, once the data is written, a version number must be assigned to the writer. Since we decided that version numbers are assigned in an incremental fashion, this step does require global synchronization and has to be serial. Moreover, since total ordering is now guaranteed and since we want to support differential updates, generating metadata for a snapshot that was assigned version number v relies on the metadata of snapshots with lower version numbers. This dependency apparently seems to require serialization too, as the metadata of the snapshots whose versions are $1 \ldots v-1$ should be generated before generating the metadata for snapshot version v.

While the serialization of version assignment is unavoidable, it is not a major concern, because this is a minor step in the write operation that generates negligible overhead compared to the rest of the write operation. However, the serialization of writing the metadata is an undesirable constraint that cancels the benefits of using a distributed metadata management scheme. It is this serialization that we want to eliminate in order to further enhance the degree of concurrency in the process of writing data.

To this end, we introduce a key concept that we refer to as *metadata forward references*. More precisely, given a snapshot version k that was successfully generated and a set of concurrent writers that were assigned versions $k+1 \ldots v$, the process of generating the metadata for snapshot version v must be able to precalculate all potential references to metadata belonging to versions $k+1 \ldots v-1$ *even though the metadata of these versions have not been written yet*, under the assumption that they will eventually be written in the future, leading to a consistent state. Considering this condition satisfied, metadata can be written in a parallel fashion as each writer can precalculate its metadata forward references individually if necessary.

However, consistency is guaranteed only if a new snapshot version v is considered as successfully generated after the metadata of all snapshots with a lower version are successfully written. This is so in order to ensure that all potential metadata forward references have been solved. Thus snapshot generation is an extremely cheap operation, as it simply involves delaying the revealing of snapshot version v to readers until the metadata of all lower snapshot versions has been written.

Advantages. Avoiding synchronization between concurrent accesses both at data and metadata level unlocks the potential to access data in a highly concurrent fashion. This approach is combined with data striping and metadata decentralization, so that concurrent accesses are physically distributed at large scale among nodes. This combination is crucial in achieving a high throughput under heavy access concurrency.

4.2.3 Consistency semantics

We adopt *linearizability* [59] as the model to reason about concurrency. Linearizability provides the illusion that each operation applied by concurrent processes appears to take effect instantaneously at some moment in time between the invocation and the completion of the operation. This provides strong consistency guarantees that enable easy reasoning about concurrency at application level.

In our case, the objects we reason about are snapshots. Readers access snapshots explicitly specified by the snapshot version. Writers do not access any explicitly specified snapshot, so we associate them to an implicit virtual snapshot, which intuitively represents the most recent view of the BLOB. The effect of all writes is guaranteed to satisfy total ordering, which in terms of linearizability, is the sequential specification of the virtual snapshot. Since linearizability does not place any constraints on the definition of the semantics of operations, we make an important decision: we define the completion of the read primitive to be the moment when its callback was invoked, as is intuitively natural, but on the other hand, we define the completion of the write primitive to be the moment when the assigned snapshot version was successfully generated, rather than the moment the corresponding callback was invoked.

Using this definitions, we show that any interleaving of reads and writes is linearizable. First observe that readers and writers never access the same snapshot. Because linearizability is a local property (any interleaving of operations on different objects is linearizable if and only if all interleavings on operations on the same object, taken separately, are linearizable), we can thus analyze reads and writes separately. An interleaving of reads is obviously linearizable. An interleaving of writes is linearizable because it satisfies the two conditions for linearizability: (1) each write has a linearization point, because the write operation is guaranteed to be atomic; and (2) the order of non-concurrent writes is preserved, because total ordering is guaranteed and therefore the effect of an already completed first write is never applied after the effect of a subsequent second write which was invoked after the completion of the first write.

In our model we have chosen to define the completion of the write primitive as the moment in time when its assigned snapshot has been generated. This has an important consequence: the callback of the write operation may be invoked *before* the the corresponding

snapshot has been generated and exposed by the system to the readers. This can potentially lead to a situation where a snapshot cannot be read immediately after it has been written, even by the same process. This does not lead to an inconsistent state, but is rather a design choice: the reader is forced to access an explicitly specified snapshot version.

This approach has a major advantage: writers don't have to wait for their updates to get generated, which enables taking decisions about future writes much earlier, greatly enhancing parallelism. This feature enhances parallelism without sacrificing the ease of use provided by strong consistency guarantees and still avoids the need for complex higher-level synchronization mechanisms that are necessary for weaker models.

The introduction of the `CLONE` and `MERGE` primitives also introduces the potential to enable efficient *semantic-based reconciliation* under concurrency [120, 141]. As argued in [5], under high contention conditions and high number of accesses per client, traditional lock-based approaches are doomed to fail in practice due to an exponential growth in the probability of deadlocks. With semantic reconciliation, concurrent clients work asynchronously in relative isolation, updating data independently and repairing occasional conflicts when they arise at semantic level, which avoids having to lock the data out while somebody else is updating it. Since the application is aware of the semantics of conflicts, in many cases conflicts can be repaired without having to abort and restart transactions, as is the case with the traditional approach.

Using `CLONE` and `MERGE`, each client can isolate its updates from the other concurrent clients and then merge back the updates in the original BLOB after repairing the conflicts. If the updates applied to a clone are undesired and need to be revoked, simply forgetting about the clone is enough. Compared to traditional approaches, which need to rollback the updates, this effectively avoids undesired effects such as cascading rollbacks, which can introduce serious performance degradation. Since both `CLONE` and `MERGE` are extremely cheap, high-performance semantic-based reconciliation can be easily implemented on top of them.

Moreover, even if the application needs transactional support in the traditional sense, the two primitives offer a solid foundation to build a highly efficient transactional manager based on *snapshot-isolation* [18].

Chapter 5

High level description

Contents

THIS chapter introduces BlobSeer, a large-scale data management service that illustrates the design principles introduced in Chapter 4. First, the architecture of BlobSeer is presented, followed by a general overview of how the reads, writes and appends work. Then, an algorithmic description is given that focuses on the versioning aspects described in Section 4.2.

5.1 Global architecture

BlobSeer consists of a series of distributed communicating processes. Figure 5.1 illustrates these processes and the interactions between them.

Clients create, read, write and append data from/to BLOBs. A large number of concurrent clients is expected that may simultaneously access the same BLOB.

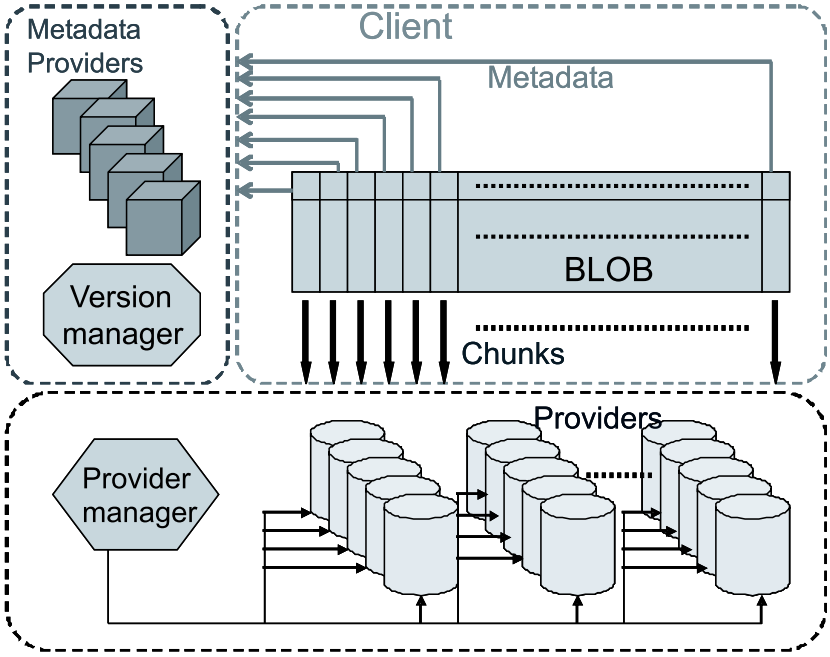


Figure 5.1: Global architecture of BlobSeer

Data (storage) providers physically store the chunks generated by appends and writes. New data providers may dynamically join and leave the system.

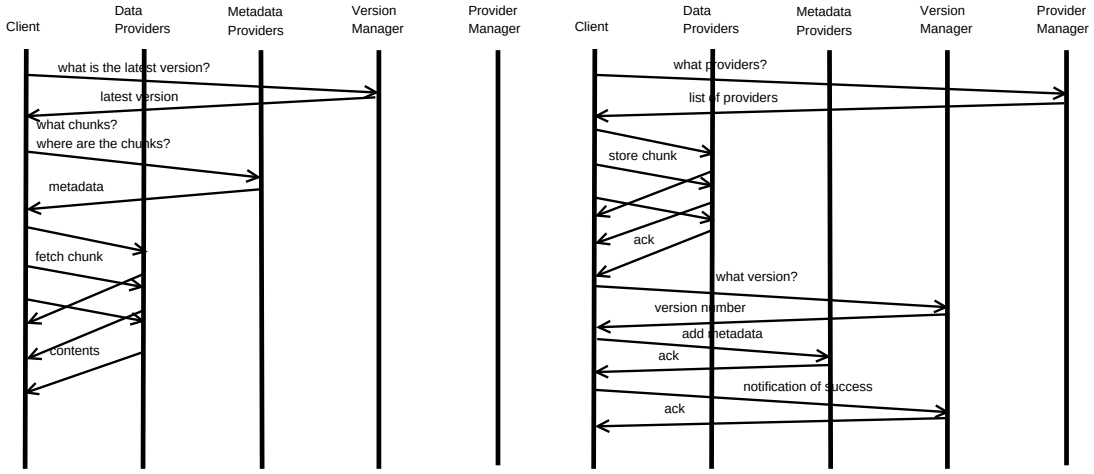
The provider manager keeps information about the available storage space and schedules the placement of newly generated chunks. It employs a configurable chunk distribution strategy to maximize the data distribution benefits with respect to the needs of the application.

Metadata (storage) providers physically store the metadata that allows identifying the chunks that make up a snapshot version. A distributed metadata management scheme is employed to enhance concurrent access to metadata.

The version manager is in charge of assigning new snapshot version numbers to writers and appenders and to reveal these new snapshots to readers. It is done so as to offer the illusion of instant snapshot generation and to satisfy the guarantees listed in Section 4.2.2.

5.2 How reads, writes and appends work

This section describes the basic versioning-oriented access primitives introduced in Section 4.2 (read, write and append) in terms of high-level interactions between the processes presented in the previous section. Processes use remote procedure calls (RPCs) [147, 156] in order to communicate among each other.



(a) Reading a subsequence (b) Writing a subsequence

Figure 5.2: Reads and Writes

Reading data. To read data (Figure 5.2(a)), the client first contacts the version manager to check if the requested snapshot version already exists in the system. If this is not the case, the read operation fails. Otherwise the client queries the metadata providers for the metadata indicating which providers store the chunks corresponding to the requested subsequence in the blob delimited by offset and size. Once these data providers have been established, the client fetches the chunks in parallel from the data providers. Note that the range specified by the read request may be unaligned to full chunks. In this case, the client requests only the relevant parts of chunks from the data providers.

Writing and appending data. To write data (Figure 5.2(b)), the client first splits the data to be written into chunks. It then contacts the provider manager and informs it about the chunks to be written. Using this information from the client, the provider manager selects a data provider for each chunk and then builds a list that is returned to the client. Having received this list, the client contacts all providers in the list in parallel and sends the corresponding chunk to each of them. As soon as a data provider receives a chunk, it reports success to the client and caches the chunk which is then asynchronously written to the disk in the background. After successful completion of this stage, the client contacts the version manager and registers its update. The version manager assigns to this update a new snapshot version v and communicates it to the client, which then generates new metadata that is "weaved" together with the old metadata such that the new snapshot v appears as a standalone entity. Finally, it notifies the version manager of success, and returns successfully to the user. At this point, the version manager takes the responsibility of eventually revealing the version v of the BLOB to the readers, which is referred to as successful generation of snapshot v.

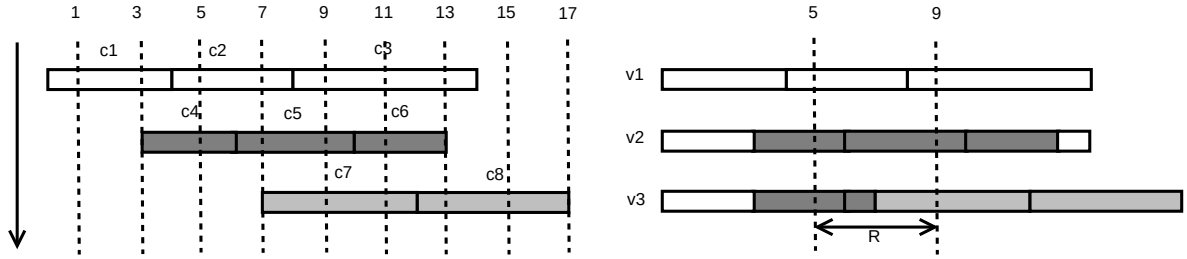


Figure 5.3: Three successive writes (left) and the corresponding snapshots (right)

5.3 Data structures

The rest of this chapter introduces a series of algorithmic descriptions that detail the interactions presented in the previous section. In order to simplify the notation, it is assumed that a single BLOB is involved in all operations and therefore the BLOB *id* is omitted from the presentation. This does not restrict the general case in any way, as all data structures presented in this section can be instantiated and maintained for each BLOB *id* independently.

Let v be a snapshot version of a BLOB and R an arbitrary subsequence in snapshot v that is delimited by *offset* and *size*. In this case, R consists of all full chunks and/or parts of chunks that lie between *offset* and *offset* + *size*, and which were written at the highest version smaller or equal to v.

For example, Figure 5.3 depicts a scenario in which an initially empty BLOB is updated three times (left side), which results in the snapshots whose composition is illustrated on the right side. For convenience, let's assume that each unit represents 1 MB. Thus, the first update (white) is an append of 14 MB in three chunks: $c1$, $c2$, $c3$ (4 MB, 4 MB, 6 MB). The second update (dark gray) is a write of 10 MB in three chunks: $c4$, $c5$, $c6$ (3 MB, 5 MB, 3 MB), starting at offset 3. Finally the last update (light gray) is a write of 10 MB starting at offset 7 in two chunks: $c7$ and $c8$ (5 MB, 5 MB).

Now let's assume a client wants to read the subsequence R delimited by $offset = 5$ and $size = 4$ of the snapshot $v = 3$ that results after all three updates are applied. In this case, the composition of R consists of the last MB of $c4$, the first MB of $c5$ and the first two MB of $c7$.

To R we associate the *descriptor map* D that fully describes the composition of R as list of entries. Each entry in D is a *chunk descriptor* of the form *(cid, coffset, csize, roffset)*, and represents either a full chunk or a part of a chunk that belongs to R: *cid* is the id that uniquely identifies the chunk in the system; *coffset* and *csize* delimit the chunk part relative to the beginning of the chunk (for a full chunk $coffset = 0$ and *csize* is the total size of the chunk); and finally *roffset* is the relative offset of the chunk part with respect to the absolute offset of R in the snapshot v.

Considering the scenario given as an example above, the associated descriptor map contains the following chunk descriptors: $(c4, 2, 1, 0), (c5, 0, 1, 1), (c7, 0, 2, 2)$.

Such descriptor maps are shared among the processes. We assume this is done through a globally shared container D_{global} that enables any process to concurrently store and retrieve descriptor maps. To anticipate the question whether this is a potential bottleneck, we show in Chapter 7 how such globally shared containers can be implemented efficiently in a distributed fashion. Each descriptor map is identified by a globally unique id. Let i_D be the id

of the descriptor map D. We denote the store operation by $D_{global} \leftarrow D_{global} \cup (i_D, D)$ and the retrieve operation by $D \leftarrow D_{global}[i_D]$.

A similar container P_{global} is used to share information about the location of chunks. Entries in this case are of the form $(cid, address)$, where *cid* is the unique id that identifies the chunk in the system and *address* the identifier of the data provider which stores *cid*.

Finally, we define H_{global} to be the history of all writes in the system that were assigned a snapshot version, regardless whether their snapshot was generated or is in the course of being generated. Entries in this history are of the form $(v, (t, o, s, i))$, where: v is the snapshot version assigned to the write; t, the total size of the snapshot *after* the update; o, the offset of the update in the snapshot; s, the size of the update; and i, the identifier of the descriptor map associated to the update. H_{global} enables global sharing of these entries, in a manner similar to D_{global}. However, rather than retrieving a single entry at a time, we require H_{global} to be able to supply a whole subsequence of entries whose versions lie between v_a and v_b in a single step. We denote this operation with $H_{global}[v_a \dots v_b]$.

Taking again the example in Figure 5.3 where white was assigned version 1, dark gray version 2 and light gray version 3, the corresponding descriptor maps are:

$$D_1 = \{(c1, 0, 4, 0), (c2, 0, 4, 4), (c3, 0, 6, 8)\}$$
$$D_2 = \{(c4, 0, 3, 3), (c5, 0, 4, 6), (c6, 0, 3, 10)\}$$
$$D_3 = \{(c7, 0, 5, 7), (c8, 0, 5, 12)\}$$

Assuming the associated globally unique identifiers of the descriptor maps are i_1, i_2 and i_3, the history of all writes contains the following entries:

$$H_{global} = \{(1, (14, 0, 14, i_1)), (2, (14, 3, 10, i_2)), (3, (17, 7, 10, i_3))\}$$

5.4 Algorithms

Using the data structures presented above, this section details the algorithms used to read, write and generate new snapshots. We omit the details of metadata management, which are presented in Chapter 6. Furthermore, in Section 4.2.1.3 we argued for the need to clone a BLOB and then eventually merge the updates performed on the clone selectively into another BLOB. Since these operations involve already existing data, they are handled at metadata level only and therefore are presented in Chapter 6 as well.

5.4.1 Learning about new snapshot versions

Before being able to read data, a client needs to find out about the versions of the snapshots that were successfully generated and are available for reading. Since total ordering is guaranteed, it is enough to learn about the version of the most recently generated snapshot, as all snapshots with lower versions are also available for reading.

The most recently generated snapshot, whose version is denoted v_g, is maintained by the version manager and is is exposed to clients through the remote procedure `RECENT`, that takes

no arguments and simply returns v_g. In order to simplify notation, we refer to a snapshot whose version is v simply as snapshot version v.

Using the BlobSeer API, at application level v_g can be obtained calling the `GET_RECENT` primitive, presented in Algorithm 1. `GET_RECENT` simply invokes `RECENT` remotely on the version manager and then invokes the *callback* supplied by the application with the result of `RECENT`. Note that v_g is not necessarily the highest version in the system. Writes using the BlobSeer API that were assigned higher versions than v_g may have successfully completed, without their snapshot being revealed to the clients yet.

Algorithm 1 Get a recent snapshot version

```
1: procedure GET_RECENT(callback)
2:     v ← invoke remotely on version manager RECENT
3:     invoke callback(v)
4: end procedure
```

In order to obtain the total size of a snapshot whose version is v, the client has to call the `GET_SIZE` primitive, described in Algorithm 2. Obviously, this version needs to be available for reading. The primitive queries H_{global} for the extra information stored about v. Once the total size t has been successfully obtained, the *callback* is invoked with t as parameter.

Algorithm 2 Get the size of a snapshot

```
1: procedure GET_SIZE(v, callback)
2:     (t, _, _, _) ← H_global[v]
3:     invoke callback(t)
4: end procedure
```

In Section 4.2.1 two additional primitives were introduced that enable learning about new versions: `SUBSCRIBE` and `UNSUBSCRIBE`. These two primitives register/unregister the interest of the invoking client to receive notifications about newly generated snapshots in the system. We omit the algorithmic description of these primitives, as they rely on publish-subscribe mechanisms, which are widely covered in the literature [43, 11]. In this case, the version manager acts as the publisher of v_g each time v_g is incremented as the result of successful generation of new snapshots.

5.4.2 Reading

Using the primitives presented above, the client can obtain information about snapshots that are available for reading can consequently can initiate a read of any subsequence of any snapshot. This is performed using the `READ` primitive, presented in Algorithm 3, which consists of the following steps:

1. The client first invokes `RECENT` on the version manager to obtain v_g. If the requested version is higher than v_g or if the requested offset and size overflow the total size of the snapshot ($offset + size > t$), then `READ` is aborted and the supplied *callback* is invoked immediately to signal failure.

2. Otherwise, the client needs to find out what parts of chunks fully cover the requested subsequence delimited by *offset* and *size* from snapshot version v and where they are stored. To this end, the primitive `GET_DESCRIPTORS`, detailed in Section 6.3.1, builds the descriptor map of the requested subsequence.

3. Once it has build the descriptor map of the subsequence, the client proceeds to fetch the necessary parts of the chunks in parallel from the data providers into the locally supplied buffer. Although not depicted explicitly in Algorithm 3, if any get operation failed, the other parallel unfinished get operations are aborted and the *callback* is invoked immediately to signal failure to the user.

4. If all these steps have successfully completed, the `READ` primitive invokes the *callback* to notify the client of success.

Algorithm 3 Read a subsequence of snapshot version v into the local buffer

1: **procedure** `READ`(v, *buffer*, *offset*, *size*, *callback*)
2: $v_g \leftarrow$ **invoke remotely** on version manager `RECENT`
3: **if** $v > v_g$ **then**
4: **invoke** *callback*(**false**)
5: **end if**
6: $(t, \_, \_, \_) \leftarrow H_{global}[v]$ ▷ t gets the total size of the snapshot v
7: **if** *offset* + *size* $> t$ **then**
8: **invoke** *callback*(**false**)
9: **end if**
10: $D \leftarrow$ `GET_DESCRIPTORS`(v, t, *offset*, *size*)
11: **for all** *(cid, coffset, csize, roffset)* $\in D$ **in parallel do**
12: *buffer*[*roffset* .. *roffset* + *csize*] $\leftarrow$ **get** subsequence [*coffset* .. *coffset* + *csize*] of chunk *cid* **from** $P_{global}[cid]$
13: **end for**
14: **invoke** *callback*(**true**)
15: **end procedure**

5.4.3 Writing and appending

The `WRITE` and `APPEND` primitives are described in Algorithm 4. It is assumed that either a default or a custom partitioning function was configured by the user to split the local buffer into chunks. This partitioning function is denoted `SPLIT` and takes a single argument: the total size of the buffer to be split into chunks. It returns a list of chunk sizes, denoted K, whose sum adds up to the total size of the buffer.

The main idea behind `WRITE` and `APPEND` can be summarized in the following steps:

1. Split the local buffer into $|K|$ chunks by using the partitioning function `SPLIT`.

2. Get a list of $|K|$ data providers, one for each chunk, from the provider manager.

3. Write all chunks in parallel to their respective data providers and build the corresponding descriptor map D relative to the *offset* of the update. Add an entry for each chunk to

Algorithm 4 Write a the content of a local buffer into the blob

```
1:  procedure WRITE(buffer, offset, size, callback)
2:      K ← SPLIT(size)
3:      P ← get |K| providers from provider manager
4:      D ← ∅
5:      for all 0 ≤ i < |K| in parallel do
6:          cid ← uniquely generated chunk id
7:          roffset ← Σ_{j=0}^{i-1} K[j]
8:          store buffer[roffset .. roffset + K[i]] as chunk cid on provider P[i]
9:          D ← D ∪ {(cid, 0, K[i], roffset)}
10:         P_global ← P_global ∪ {(cid, P[i])}
11:     end for
12:     i_D ← uniquely generated id
13:     D_global ← D_global ∪ (i_D, D)
14:     (v_a, v_g) ← invoke remotely on version manager ASSIGN_VERSION_TO_WRITE(offset, size,
    i_D)
15:     BUILD_METADATA(v_a, v_g, D)
16:     invoke remotely on version manager COMPLETE(v_a)
17:     invoke callback(v_a)
18: end procedure
```

P_{global}. Although not depicted explicitly in Algorithm 4, if any store operation fails, all other parallel store operations are aborted and `WRITE` invokes the *callback* immediately to return failure.

4. Add the descriptor map D to D_{global}.

5. Ask the version manager to assign a new version number v_a for the update (`ASSIGN_VERSION_TO_WRITE`), then write the corresponding metadata (`BUILD_METADATA`) and notify the version manager that the operation succeeded (`COMPLETE`). Once the version manager receives this notification, it can generate the snapshot v_a at its discretion.

The last step requires some closer consideration. It is the responsibility of the version manager to assign new snapshot versions, which can be requested by clients through the `ASSIGN_VERSION_TO_WRITE` remote procedure call. Since we aim to guarantee total ordering, new versions are assigned in increasing order. To this end, the version manager keeps only the latest assigned version, denoted v_a, which is atomically incremented for each new write request. Moreover, the version manager also takes the responsibility to add all necessary information about v_a (the total size of snapshot version v_a, *offset*, *size*, i_D) to the history of all writes H_{global}. This process is detailed in Algorithm 5.

In the case of append, *offset* is not specified explicitly, it is implicitly equal to the total size stored by $H_{global}[v_a - 1]$. Since this is the only difference from writes, we do not develop the pseudo-code for `APPEND` and `ASSIGN_VERSION_TO_APPEND` explicitly.

Both v_g and v_a are reported back to the client. Note that since each writer publishes D, it is possible to establish the composition of any snapshot version by simply going backwards in the history of writes H_{global} and analyzing corresponding descriptor maps $D_{global}[i]$. How-

Algorithm 5 Assign a snapshot version to a write

```
1:  function ASSIGN_VERSION_TO_WRITE(offset, size, i_D)
2:      (t_a, _, _, _) ← H_global[v_a]
3:      if offset + size > t_a then
4:          t_a ← offset + size
5:      end if
6:      v_a ← v_a + 1
7:      H_global ← H_global ∪ {(v_a, (t_a, offset, size, i_D))}
8:      Pending ← Pending ∪ {v_a}
9:      return (v_a, v_g)
10: end function
```

ever, such an approach is unfeasible as it degrades read performance as more writes occur in the BLOB. As a consequence, more elaborate metadata structures need to be maintained.

The `BUILD_METADATA` function, detailed in Section 6.3.2, is responsible to generate such new metadata that both references the descriptor map D and the metadata of previous snapshot versions, such as to provide the illusion of a fully independent snapshot version v_a, yet keep the performance levels of querying metadata close to constant, regardless of how many writes occurred in the system, that is, regardless of how large v_a is. Moreover, it is designed to support *metadata forward references*, introduced in Section 4.2.2. This avoids having to wait for other concurrent writers to write their metadata first, thus enabling a high-throughput under write-intensive scenarios. These aspects related to metadata management are discussed in more detail in Chapter 6.

After the client finished writing the metadata, it notifies the version manager of success by invoking `COMPLETE` remotely on the version manager and finally invokes the *callback* with the assigned version v_a as its parameter. At this point, the write operation completed from the client point of view and it is the responsibility of the version manager to publish snapshot v_a.

5.4.4 Generating new snapshot versions

The version manager has to generate new snapshots under the guarantees mentioned in Section 4.2.1: liveness, total ordering and atomicity.

In order to achieve this, the version manager holds two sets: *Pending* and *Completed*. The *Pending* set holds all versions for which the metadata build process is "in progress". More specifically, these are the versions that were assigned to clients through either `ASSIGN_VERSION_TO_WRITE` or `ASSIGN_VERSION_TO_APPEND`, but for which the client didn't confirm completion yet by invoking `COMPLETE`. The *Completed* set holds all versions that "are ready for generation". More specifically, these are the versions for which a notification has been received but the corresponding snapshot has not been revealed yet to the readers (i.e., v_g is lower than all versions in *Completed*).

Each time a client requests a new version, v_a is incremented and added to the *Pending* set (Line 8 of Algorithm 5). Once a notification of completion for v_a is received, `COMPLETE`, presented in Algorithm 6 is executed on the version manager. Naturally, v_a is moved from *Pending* into *Completed* to mark that v_a is ready to be generated. If v_a is exactly $v_g + 1$, the

version manager generates $v_g + 1$, increments v_g and eliminates it from *Completed*. This step can lead to a cascading effect where multiple successive versions, higher that v_a, that were reported by clients as completed before v_a can now themselves be published. This is why `COMPLETE` iteratively tries to perform the step until no more completed versions can be published anymore.

Algorithm 6 Complete the write or append

```
1: procedure COMPLETE(v_a)
2:     Pending ← Pending \ {v_a}
3:     Completed ← Completed ∪ {v_a}
4:     while v_g + 1 ∈ Completed do
5:         v_g ← v_g + 1
6:         Completed ← Completed \ {v_g}
7:     end while
8: end procedure
```

Using this approach satisfies the guarantees mentioned in Section 4.2.1:

Total ordering implies snapshot v_a cannot be generated unless the metadata of all snapshots labeled with a lower version have been fully constructed, which translates into the following condition: all writers that were assigned a version number $v_i < v_a$ notified the version manager of completion. Since v_g is incremented only as long as a notification of completion for $v_g + 1$ was received, this property is satisfied.

The liveness condition is satisfied assuming that writers do not take forever to to notify the version manager of completion, because eventually v_g will be incremented to reach v_a. If a pending writer fails to perform the notification in a predefined amount of time, then a failure can be assumed and the process of building the metadata can be delegated to any arbitrary process (for example to any metadata provider). This is possible because both the chunks and their corresponding descriptor map were successfully written *before* the client failed (otherwise the client would not have requested a snapshot version for its update in the first place). The situation where the client failed after writing the metadata partially or fully does not lead to inconsistencies: the backup process does not need to be aware of this and can safely overwrite any previously written metadata, as the snapshot has not been revealed yet to the readers.

Atomicity is satisfied because the clients can learn about new snapshot versions only from the version manager. Since v_g is atomically incremented on the version manager, a call to `GET_RECENT` that returns a higher version than was previously known by the client corresponds from the client's point of view to an instant appearance of a fully independent snapshot in the system, without any exposure to inconsistent transitory states.

5.5 Example

We consider the same example previously presented in Figure 5.3 in order to illustrate the algorithms presented above. For convenience, we repeat it again in this section as Figure 5.4.

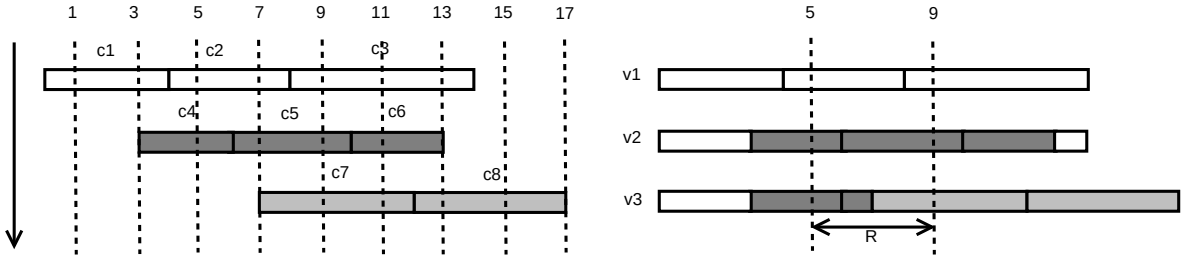


Figure 5.4: Three successive writes (left) and the corresponding snapshots (right)

The scenario we assume is the following: two concurrent writers, light gray and dark gray start writing at the same time in a blob whose latest snapshot version is 1 and which consists of three chunks, $c1, c2$ and $c3$, that add up to a total of 14 MB (white). In this initial state, assuming the corresponding descriptor map is labeled i_1, we have:

$$\begin{aligned} H_{global} &= \{(1, (14, 0, 14, i_1))\} \\ v_g = v_a &= 1 \\ Pending = Completed &= \emptyset. \end{aligned}$$

After obtaining a list of data providers able to store the chunks, each writer proceeds to send the chunks in parallel to the corresponding data providers and add the corresponding chunk descriptors to the local descriptor map D associated to the update. Note that the writers perform these operations concurrently in full isolation, without any need for synchronization among each other.

Once these steps are complete, each writer generates a globally unique id for D. In order to be consistent with the notations used in Section 5.3 for the same example, we assume the globally unique id of D for dark gray is i_2, while the id for light gray is i_3. Each of the writers independently adds its local D to D_{global} and asks the version manager to assign a version to their update by invoking `ASSIGN_VERSION_TO_WRITE` remotely.

For the purpose of this example, dark gray is first. Thus the version manager increments v_a to 2, adds the corresponding entry in the history of writes and adds v_a to $Pending$. At this point,

$$\begin{aligned} H_{global} &= \{(1, (14, 0, 14, i_1)), (2, (14, 3, 10, i_2))\} \\ v_g &= 1 \\ v_a &= 2 \\ Pending &= \{2\} \\ Completed &= \emptyset \end{aligned}$$

Next, light gray asks for a version and is assigned $v_a = 3$. This leads to the following

state:

$$\begin{aligned} H_{global} &= \{(1,(14,0,14,i_1)),(2,(14,3,10,i_2)),(3,(17,7,10,i_3))\} \\ v_g &= 1 \\ v_a &= 3 \\ Pending &= \{2,3\} \\ Completed &= \emptyset \end{aligned}$$

Both writers build the metadata associated to their update concurrently by calling `BUILD_METADATA`. Again note that there is no need for synchronization. The mechanisms that make this possible are discussed in more detail in Chapter 6.

In this example, we assume light gray is faster and finishes writing the metadata before dark gray, thus being the first to invoke `COMPLETE` on the version manager. Since the metadata of light gray depends on the metadata of dark gray (dark gray was assigned a lower version), the snapshot corresponding to light gray cannot be generated yet, thus v_g remains unchanged, but light gray is marked as completed. The new state is:

$$\begin{aligned} H_{global} &= \{(1,(14,0,14,i_1)),(2,(14,3,10,i_2)),(3,(17,7,10,i_3))\} \\ v_g &= 1 \\ v_a &= 3 \\ Pending &= \{2\} \\ Completed &= \{3\} \end{aligned}$$

If at this point any client wants to read the subsequence R, delimited by $offset = 5$ and $size = 4$, the only snapshot version available to read from is 1, corresponding to white. Any attempt to read from a higher snapshot version fails, as `RECENT` will return $v_g = 1$.

At some point, dark gray finishes writing the metadata corresponding to its update and calls `COMPLETE` on the version manager, which in turn marks dark gray as completed. Since $v_g = 1$ and both version 2 and 3 belong to $Completed$, v_g is incremented twice and $Completed$ is emptied, leading to the following final state:

$$\begin{aligned} H_{global} &= \{(1,(14,0,14,i_1)),(2,(14,3,10,i_2)),(3,(17,7,10,i_3))\} \\ v_g &= 3 \\ v_a &= 3 \\ Pending &= Completed = \emptyset \end{aligned}$$

Now, a client that invokes the `GET_RECENT` and `GET_SIZE` primitives obtains 3 and 17 respectively, thus a read on the same subsequence R for snapshot version 3 is successful this time. First, the client generates the descriptor map D corresponding to R by calling `GET_DESCRIPTORS`, which leads to $D = \{(c4,2,1,0),(c5,0,1,1),(c7,0,2,2)\}$. Having obtained D, the client proceeds to fetch all parts of the chunks in parallel from the corresponding data providers to assemble R in the locally supplied buffer and then notifies the user of success by invoking the callback.

Chapter 6

Metadata management

Contents

METADATA has the role of organizing the chunks in the system in such way as to offer the illusion of complete and independent snapshot versions despite writing only differential updates. This chapter describes the data structures and algorithms that enable efficient management of metadata in order to fill this role.

6.1 General considerations

In Chapter 5 we have introduced a series of versioning-oriented algorithms that generate a new snapshot of the whole BLOB for each fine-grain update, while exposing all such snapshots in a totally ordered fashion to the readers. We have also introduced a series of basic globally shared metadata structures that enable establishing the composition of snapshots in terms of chunks, and the location where those chunks are stored.

However, based on that metadata only, the complexity of establishing the composition of snapshots is linear with respect to the number of updates that occur on the BLOB. This leads to a read access performance degradation that is not acceptable. Thus, there is a need

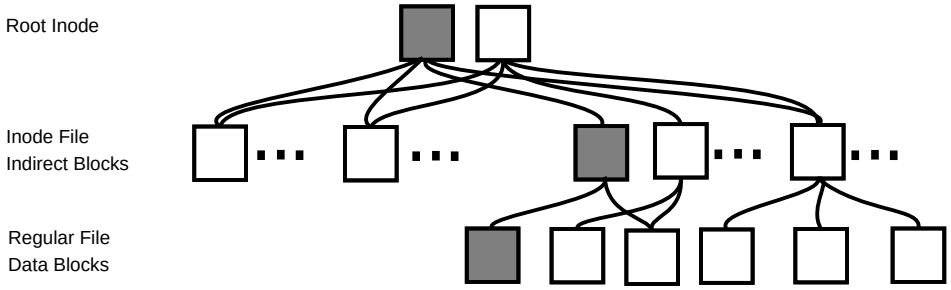


Figure 6.1: Shadowing: updates propagate to the root inode

to design and maintain additional metadata structures that are able to keep the level of read performance from the BLOB the same, regardless how many times it was updated.

In this context, balanced trees (in particular B-trees [137, 54]) are used by most file systems to maintain the composition of files and directories. Unlike traditional indirect blocks [91], B-trees offer worst-case logarithmic-time key-search, insert, and remove. This property makes B-trees highly appealing in our context. However, in our context each update to the BLOB generates a new snapshot that offers the illusion of an independent object. Using a B-tree for each snapshot is thus not feasible due to space constraints.

In order to deal with this issue, several versioning file systems, such as [58, 62], that support snapshots rely on a powerful mechanism called *shadowing*. Shadowing means to build snapshots by using copy-on-write for each on-disk data block that needs to be updated. Since after an update the on-disk location of some data blocks has changed, the metadata of the filesystem (traditionally organized into an inode hierarchy) has to be updated to reflect this change as well. With shadowing this is performed by using copy-on-write for inodes in a bottom-up manner, starting from the ones that point to the data blocks and going up towards the root inode. An example is shown in Figure 6.1, where the initial inode structure of a filesystem (white) points to the data blocks of two files, each of which is described by an indirect inode block. Assuming that the leftmost data block needs to be changed, shadowing uses copy-on-write for all inodes on the path from the data block to the root inode (gray) in order to build a new snapshot of the filesystem.

Adapting B-trees to support shadowing however is not trivial. Traditional B-tree maintenance methods [97, 85, 84] try to minimize the cost of rebalancing (necessary to guarantee logarithmic operations) by keeping the changes as local as possible in order to avoid having to reorganize the tree nodes all the way up to the root. In order to do so, they introduce additional references among tree nodes that aid in improving locality. While these methods are highly efficient for simple B-trees, they cannot be easily adapted to support shadowing, because the extra references can cause "chain-reactions", i.e. a simple change ultimately leads to the whole B-tree having to be copied.

Recent work [131] aims at adapting B-trees for shadowing. A series of algorithms are presented that organize the B-tree in such way that references do not cause "chain-reactions" and thus large parts of the B-tree can be shared between snapshots. Moreover, the proposed algorithms also aim to be efficient when the operation on the B-tree are performed concurrently. In order to do so, lock-coupling [16] (or crabbing) is used as a synchronization mechanism (i.e. locking children before unlocking the parent). This ensures the validity of a tree

path that a thread is traversing without prelocking the entire path and is deadlock-free. The authors claim to attain performance levels close to regular B-trees even under concurrency, despite removing the extra references.

This approach is apparently feasible to adapt for use in our context, because it introduces support for both shadowing and concurrent access, while still inheriting the worst-case logarithmic time guarantee. However, in our context, as mentioned in Section 4.1.3, the metadata grows to huge sizes and needs to be maintained in a distributed fashion for scalability purposes. This in turn means that a distributed locking scheme would be necessary to make this approach work, which is known to be a problematic issue.

Moreover, a locking scheme implies that access to metadata is synchronized, which limits the potential of attaining efficient write/write concurrency. As explained in Section 4.2.2, ideally synchronization at metadata level should be avoided altogether by means of *metadata forward references*. Essentially this means to organize the references in the metadata of a snapshot in such way that it is possible to "guess" the references to the metadata of lower snapshot versions even though *it has not been written yet*, under the assumption that it will be eventually written and thus a consistent state is obtained.

To this end, we propose in this chapter a set of metadata structures and algorithms oriented towards shadowing that overcome the limitations of B-trees, while still offering worst-case logarithmic access times.

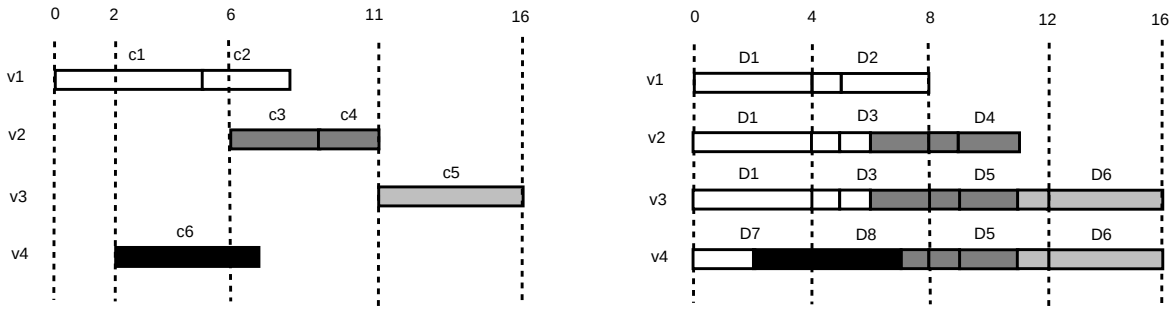
6.2 Data structures

We organize metadata as a *distributed segment tree* [174], and associate one to each snapshot version of a given BLOB. A segment tree is a binary tree in which each node is associated to a subsequence of a given snapshot version v_i of the BLOB, delimited by *offset* and *size*. In order to simplify the notation, we denote *offset* with x_i and *offset + size* with y_i and refer to the subsequence delimited by *offset* and *size* as segment $[x_i, y_i]$. Thus, a node is uniquely identified by the pair $(v_i, [x_i, y_i])$. We refer to the association between the subsequence and the node simply as the node *covers* $(v_i, [x_i, y_i])$ (or even shorter, the node *covers* $[x_i, y_i]$).

For each node that is not a leaf, the left child covers the left half of the parent's segment, and the right child covers the right half of the parent's segment. These two children are $(v_{li}, [x_i, (x_i + y_i)/2])$ and $(v_{ri}, [(x_i + y_i)/2, y_i])$. Note that the snapshot version of the left child v_{li} and right child v_{ri} must not necessarily be the same as v_i. This enables the segment trees of different snapshot versions to share whole subtrees among each other. The root of the tree is associated the minimal segment $[x_i, y_i]$ such that it covers the whole snapshot v_i. Considering t_i the total size of the snapshot v_i, the associated root covers $[0, r_i]$, where r_i is the smallest power of two greater than t_i.

A node that is a leaf, covers a segment whose length is *lsize*, a fixed size that is a power of two. Each leaf holds the descriptor map D_i, relative to x_i, that makes up the subsequence. In order to limit fragmentation, the descriptor map is restricted to hold not more that k chunk descriptors, which we call its *fragmentation threshold*.

Thus, nodes are fully represented by $(key, value)$ pairs, where $key = (v_i, [x_i, y_i])$ and $value = (v_{li}, v_{ri}, D_i)$. For leaves v_{li} and v_{ri} are not defined, while $D_i \neq \emptyset$. For inner nodes, v_{li} and v_{ri} hold the version of the left and right child, while $D_i = \emptyset$.



(a) History of writes

(b) Descriptor maps of the leaf nodes

Figure 6.2: Four writes/appends (left) and the corresponding composition of the leaves when $lsize = 4$ (right)

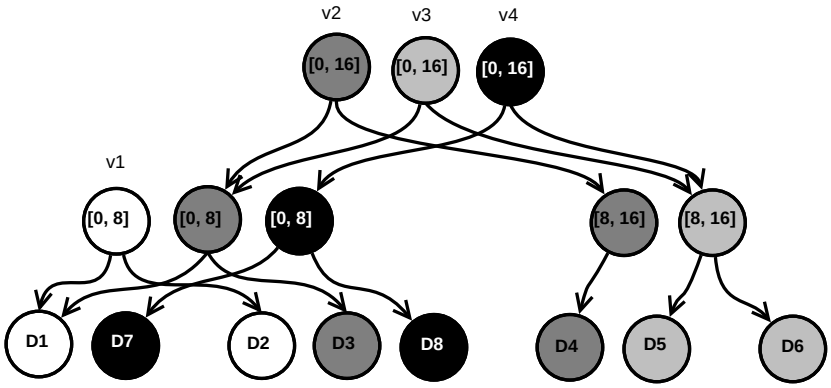


Figure 6.3: Segment tree: leaves are labeled with the descriptor maps, inner nodes with the segments they cover

Figure 6.2 depicts an example of four consecutive updates that generated four snapshots $v_1..v_4$, whose version numbers are 1..4. The updates are represented in Figure 6.2(a): each written chunk is a rectangle, chunks from the same update have the same color, and the shift on the X-axis represents the absolute offset in the blob. v_1 and v_3 correspond to appends, while v_2 and v_4 correspond to writes.

This series of appends and writes results in the snapshots presented in Figure 6.2(b). To each of these snapshots, a segment tree is associated, for which $lsize = 4$. The leaves of the segment trees thus cover the disjoint segments $[0, 4]$, $[4, 8]$, $[8, 12]$, $[12, 16]$. Some of the leaves are shared, with a total of 8 distinct leaves, labeled $D_1..D_8$ after the descriptor maps that describe their composition.

Finally, the full segment trees are depicted in Figure 6.3, where the inner nodes are labeled with the segment they cover and their links to the left and right child are represented as arrows. Notice how entire subtrees are shared among segment trees: for example, the right child of the root of black $(v_4, [0, 16])$ is light gray $(v_3, [8, 16])$.

All tree nodes associated to the BLOB are stored in the globally shared container $Nodes_{global}$ as $(key, value)$ pairs. We denote the store operation by $Nodes_{global} \leftarrow Nodes_{global} \cup (key, value)$ and the retrieve operation by $Nodes_{global} \leftarrow Nodes_{global}[key]$. Details about how

to implement such containers efficiently in a distributed fashion are given in Chapter 7.

6.3 Algorithms

6.3.1 Obtaining the descriptor map for a given subsequence

In order to read a subsequence R delimited by *offset* and *size* from the snapshot version v, we introduced the `READ` procedure in Section 5.4.2, which relies on the `GET_DESCRIPTORS` primitive to determine the descriptor map D corresponding to R before fetching the actual chunks from the data providers.

Algorithm 7 Get the descriptor map for a given subsequence

```
1:  function GET_DESCRIPTORS(v, offset, size)
2:      D ← ∅
3:      Q ← ROOT(v)
4:      while Q ≠ ∅ do
5:          ((v_i, [x_i, y_i]), (v_li, v_ri, D_i)) ← extract node from Q
6:          if D_i ≠ ∅ then                    ▷ if it is a leaf, extract the chunk descriptors
7:              D ← D ∪ INTERSECT(D_i, x_i, [x_i, y_i] ∩ [offset, offset + size], offset)
8:          else                               ▷ if it is an inner node, find the children that cover the range
9:              if [offset, offset + size] ∩ [x_i, (x_i + y_i)/2] ≠ ∅ then
10:                 Q ← Q ∪ Nodes_global[(v_li, [x_i, (x_i + y_i)/2])]
11:             end if
12:             if [offset, offset + size] ∩ [x_i + y_i/2, y_i] ≠ ∅ then
13:                 Q ← Q ∪ Nodes_global[(v_ri, [(x_i + y_i)/2, y_i])]
14:             end if
15:         end if
16:     end while
17:     return D
18: end function
```

The `GET_DESCRIPTORS` function, presented in Algorithm 7, is responsible to construct D from the segment tree associated to the snapshot. This is achieved by walking the segment tree in a top-down manner, starting from the root and reaching towards the leaves that intersect R. Once such a leaf is found, its chunk descriptors from D_i that are part of R are extracted and added to D.

The chunk descriptors of the leaf cannot be directly added to D, because the offsets in D_i are relative to x_i, which is the left end of the segment covered by the leaf, while the offsets in D need to be relative to *offset*, which is the parameter supplied to `READ`.

For this reason, the offsets of the extracted chunk descriptors need to shifted accordingly. This is performed by the `INTERSECT` function, presented in Algorithm 8.

`INTERSECT` has four arguments: the first two arguments determine the source descriptor map D_s where to extract the chunk descriptors from, and the offset x_s to which its chunk descriptors are relative. The third argument is the segment $[x_i, y_i]$ that delimits R, and finally the fourth argument is the offset x_d to which the chunk descriptors of the resulting destination descriptor map D_d are relative to.

Algorithm 8 Intersect a chunk map with a given segment

```
1: function INTERSECT(D_s, x_s, [x_i, y_i], x_d)
2:     D_d ← ∅
3:     ro_d ← max(x_s − x_d, 0)
4:     for all (cid_s, co_s, cs_s, ro_s) ∈ D_s such that [x_i, y_i] ∩ [x_s + ro_s, x_s + ro_s + cs_s] ≠ ∅ do
5:         co_d ← co_s + max(x_i − x_s − ro_s, 0)
6:         cs_d ← min(y_i − max(x_i, x_s + ro_s), cs_s)
7:         D_d ← D_d ∪ {(cid_s, co_d, cs_d, ro_d)}
8:     end for
9:     return D_d
10: end function
```

In order to illustrate how INTERSECT works, let's take again the example presented in Figure 6.2. Assuming R is delimited by $x_i = 3$ and $y_i = 7$ and the source descriptor map is D_3, we have $x_s = 4$ and $x_d = 3$. All three chunk descriptors $(c_1, 4, 1, 0)$, $(c_2, 0, 1, 1)$ and $(c_3, 0, 2, 2)$ belonging to D_3 intersect R. $x_s - x_d = 1$, thus the relative offsets are incremented by 1. Moreover, only the first unit of c_3 is part of R, such that the chunk size of the new chunk descriptor gets adjusted accordingly: $\min(7 - \max(3, 4 + 2), 2) = 1$. Thus, in the end D_d contains the following chunk descriptors: $(c_1, 4, 1, 1)$, $(c_2, 0, 1, 2)$ and $(c_3, 0, 1, 3)$.

Once all chunk descriptors from all the leaves that intersect R have been extracted, adjusted to *offset* and added to D, GET_DESCRIPTORS returns D as the final result.

6.3.2 Building the metadata of new snapshots

Once the version manager assigned a new version v_a for the update, the segment tree construction can begin. The BUILD_METADATA procedure, described in Algorithm 9 is responsible for that, starting from the following parameters: v_a, the assigned snapshot version, v_g, the version of a recently generated snapshot and D, the descriptor map of the update corresponding to v_a. Note that v_g is simply the value returned by ASSIGN_VERSION_TO_WRITE or ASSIGN_VERSION_TO_APPEND. It is possible that the version manager has published higher versions by the time BUILD_METADATA is called.

The segment tree construction is performed in a bottom-up manner, starting from the leaves and working towards the root. It consists of two stages. In the first stage, the leaves corresponding to the update are built, while in the second stage the inner nodes and the root are built. Only the subtree whose inner nodes cover at least one of the new leaves is build. Any inner node which does not cover at least one of the new leaves is shared with the most recent segment tree assigned a lower version $v_i < v_a$, which has built that inner node. This way, whole subtrees are shared among versions without breaking total ordering.

6.3.2.1 Building the leaves

First of all, the set of new leaves corresponding to the update of v_a, whose offset is o_a and size s_a, must be build. A leaf corresponds to the update if its segment $[x_i, x_i + lsize]$ covers $[o_a, o_a + s_a]$. For each such leaf, the corresponding descriptor map D_i, relative to x_i must be calculated. This is performed by the LEAF function, presented in Algorithm 10.

Algorithm 9 Build the metadata for a given snapshot version

procedure BUILD_METADATA(v_a, v_g, D)
  $W \leftarrow H_{global}[v_g \dots v_a]$
  $(t_a, o_a, s_a, \_) \leftarrow W[v_a]$
  $x_i \leftarrow \lfloor o_a / lsize \rfloor \times lsize$ ▷ largest multiple of $lsize$ smaller or equal to offset o_a
  $Q \leftarrow \emptyset$
  repeat ▷ determine all leaves that cover the written chunks
    $Q \leftarrow Q \cup \{((v_a, [x_i, x_i + lsize]), (0, 0, \texttt{LEAF}(v_a, v_g, [x_i, x_i + lsize], D, W)))\}$
    $x_i \leftarrow x_i + lsize$
  until $x_i + lsize < o_a + s_a$
  $T \leftarrow \emptyset$
  while $Q \neq \emptyset$ **do** ▷ build inner nodes in bottom-up fashion
    $((v_i, [x_i, y_i]), (v_{li}, v_{ri}, D_i)) \leftarrow$ extract any node from Q
    $T \leftarrow T \cup ((v_i, [x_i, y_i]), (v_{li}, v_{ri}, D_i))$
    if $(v_i, [x_i, y_i]) \neq \texttt{ROOT}(v_a)$ **then**
      if $(v_i, [x_i, y_i])$ has a right sibling **then**
        $(x_s, y_s) \leftarrow (y_i, 2 \times y_i - x_i)$ ▷ $[x_s, y_s]$ is the segment covered by the sibling
      else ▷ $(v_i, [x_i, y_i])$ has a left sibling
        $(x_s, y_s) \leftarrow (2 \times x_i - y_i, x_i)$
      end if
      $v_s \leftarrow v_a$ ▷ v_s is the version of the sibling, initially assumed v_a
      if $\exists((v_s, [x_s, y_s]), (v_{lj}, v_{rj}, D_j)) \in Q$ **then** ▷ sibling is in Q and has version v_a
        $Q \leftarrow Q \setminus \{((v_s, [x_s, y_s]), (v_{lj}, v_{rj}, D_j))\}$ ▷ move sibling to T
        $T \leftarrow T \cup \{((v_s, [x_s, y_s]), (v_{lj}, v_{rj}, D_j))\}$
      else ▷ sibling is not in Q, it belongs to a lower version
        $(v_s, (t_s, o_s, s_s, \_)) \leftarrow (v_a - 1, W[v_a - 1])$
        while $v_s > v_g$ **and** $[o_s, o_s + s_s] \cap [x_i, y_i] = \emptyset$ **do** ▷ determine v_s
          $v_s \leftarrow v_s - 1$
          $(t_s, o_s, s_s, \_) \leftarrow W[v_s]$
        end while
      end if
      if $x_i < x_s$ **then**
        $Q \leftarrow Q \cup \{((v_i, [x_i, y_s]), (v_i, v_s, \emptyset))\}$ ▷ add parent to Q
      else
        $Q \leftarrow Q \cup \{((v_i, [x_s, y_i]), (v_s, v_i, \emptyset))\}$
      end if
    end if
  end while
  $Nodes_{global} \leftarrow Nodes_{global} \cup T$
end procedure

Algorithm 10 Build the descriptor map for a given leaf

1: **function** LEAF($v_a, v_g, [x_i, y_i], D_a, W$)
2: $(\_, o_a, s_a, \_) \leftarrow W[v_a]$
3: $(t_g, o_g, \_, \_) \leftarrow W[v_g]$
4: $D_i \leftarrow$ INTERSECT$(D_a, o_a, [o_a, o_a + s_a] \cap [x_i, y_i], x_i)$
5: $v_j \leftarrow v_a - 1$
6: $Reminder \leftarrow [x_i, y_i] \setminus [o_a, o_a + s_a]$
7: **while** $v_j > v_g$ and $Reminder \neq \emptyset$ **do**
8: $(t_j, o_j, s_j, i_j) \leftarrow W[v_j]$
9: **for all** $[x_j, y_j] \in Reminder \cap [o_j, o_j + s_j]$ **do**
10: $D_i \leftarrow D_i \cup$ INTERSECT$(D_{global}[i_j], o_j, [x_j, y_j], x_i)$
11: $Reminder \leftarrow Reminder \setminus [x_j, y_j]$
12: **end for**
13: $v_j \leftarrow v_j - 1$
14: **end while**
15: **if** $x_i < t_g$ and $Reminder \neq \emptyset$ **then**
16: $((v_j, [x_j, y_j]), (v_{lj}, v_{rj}, D_j)) \leftarrow ROOT(v_g)$
17: **while** $x_j \neq x_i$ and $y_j \neq y_i$ **do**
18: **if** $x_j < x_i$ **then**
19: $x_j \leftarrow x_j + lsize$
20: $(v_{lj}, v_{rj}, D_j) \leftarrow Nodes_{global}[(v_{lj}, [x_j, y_j])]$
21: **else**
22: $y_j \leftarrow y_j - lsize$
23: $(v_{lj}, v_{rj}, D_j) \leftarrow Nodes_{global}[(v_{rj}, [x_j, y_j])]$
24: **end if**
25: **end while**
26: $D_i \leftarrow D_i \cup$ INTERSECT$(D_j, x_i, Reminder \cap [x_i, y_i], x_i)$
27: **end if**
28: **if** $|D_i| > k$ **then**
29: $D_i \leftarrow$ DEFRAGMENT(D_i)
30: **end if**
31: **return** D_i
32: **end function**

In order to obtain D_i, the LEAF function must extract the chunk descriptors of D that intersect with the segment of the leaf, adjusting their relative offsets from the original offset o_a of D to x_i, the offset of the leaf.

Since the leaf may not be fully covered by $[o_a, o_a + s_a]$, the remaining part of the leaf, denoted *Reminder*, may cover chunks belonging to snapshot versions lower than v_a. If the snapshot $v_a - 1$ has already been generated, that is, $v_g = v_a - 1$, extracting the chunk descriptors that fill *Reminder* can be performed directly from the descriptor map of the leaf that covers $[x_i, y_i]$ in the segment tree of the snapshot $v_a - 1$.

If $v_g < v_a - 1$, then the status of the metadata for all concurrent writes, with assigned version v_j such that $v_g < v_j < v_a$, is uncertain. Therefore, the segment tree of snapshot $v_a - 1$ cannot be relied upon to fill *Reminder*, as it may not have been generated yet.

However, since a write which was assigned a snapshot version v_j requested a version *after* it added to D_{global} its full descriptor map D_j, the *Reminder* of the leaf can be gradually filled by working backwards in the history of writes starting from $v_a - 1$ and extracting the chunk descriptors that overlap with the leaf, while adjusting the relative offset to x_i. This step might seem costly, but we need to observe that D_{global} has to be queried to obtain D_j *only if Reminder* intersects with the update of v_j, which in practice is rarely the case. Obviously, once v_g has been reached, the process stops, because the leaf of v_g can be consulted directly.

At this point, D_i has been successfully generated. It is however possible that overlapping updates have fragmented the segment $[x_i, y_i]$ heavily, such that D_i contains a lot of chunk descriptors. This is turn reduces access performance, because many small parts of chunks have to be fetched. For this reason, if the number of chunk descriptors goes beyond the fragmentation threshold, that is, $|D_i| > k$, a healing mechanism for the leaf is employed. More precisely, the DEFRAGMENT primitive, which is responsible for this task, reads the whole range $[x_i, y_i]$ and then applies the SPLIT primitive to obtain less then k chunks. These chunks are written back to the data providers and D_i is reset to contain their corresponding chunk descriptors. This effectively reorganizes D_i to hold less than k chunk descriptors.

6.3.2.2 Building the inner nodes

Having generated the set of leaves for the snapshot v_a, BUILD_METADATA proceeds to build the inner nodes of the segment tree, up towards the root. This is an iterative process: starting from the set of leaves Q, any two siblings for which at least one of them belongs to Q are combined to build the parent, which in turn is eventually combined, up to the point when the root itself is obtained and Q becomes empty.

In order to find two siblings that can be combined, an arbitrary tree node $((v_a, [x_i, y_i]), (v_{li}, v_{ri}, D_i))$ is extracted from Q and its sibling is determined by calculation. This tree node is either the left child of its parent and thus it has a right sibling or it is the right child of its parent and thus has a left sibling. In either case, the segment $[x_s, y_s]$ covered by the sibling can be easily calculated. Thus, the only missing information in order to completely determine the sibling is its version v_s. This version however needs more closer attention. If the sibling belongs itself to Q, then $v_s = v_a$. Otherwise, v_s is the version assigned to the most recent writer for which $v_s < v_a$ and the corresponding update of v_s intersects with $[x_s, y_s]$, which means the sibling was generated or is in the process of being generated by that writer. Once v_s is established, both versions of the children are available and thus

the parent can be generated.

Since the sibling may correspond to a lower snapshot version whose metadata is being concurrently generated, it is unknown whether the sibling was indeed generated yet. Thus the reference of the parent to it may be a potential *metadata forward reference*. Since snapshots are revealed to the readers only after the metadata of all lower version snapshots was written, the segment tree associated to v_a will be consistent at the time snapshot v_a is revealed to the readers.

In any of the two cases, both the node (which belongs to Q) and its sibling (if it belongs to Q) are moved from Q to T, which holds the set of all tree nodes belonging to v_a that have been successfully combined. Once the root of the segment tree has been generated, T holds the whole subtree corresponding to v_a and is committed to $Nodes_{global}$. At this point the metadata build process has successfully completed and `BUILD_METADATA` returns.

6.3.3 Cloning and merging

In Section 4.2.1.3 we introduced two special primitives: `CLONE` and `MERGE`. The `CLONE` primitive is used to create a new BLOB whose initial snapshot version duplicates the content of a given snapshot of an already existing BLOB. `MERGE` is used to write a region of a specified snapshot version of a BLOB into another BLOB. It is typically used in conjunction with `CLONE` to isolate updates to a BLOB and then selectively merge them back later in the original BLOB once all semantic conflicts have been eliminated.

We describe here how to perform these operations with minimal overhead, relying on metadata manipulation alone. In order to do so, we extend the idea of physically sharing the unmodified content between snapshots of the same BLOB to physically sharing the unmodified content between snapshots of different BLOBs as well.

So far, the *id* of the BLOB was omitted from the algorithmic notation for simplification purposes, because the same BLOB was always concerned. However, every data structure presented so far is implicitly maintained for each BLOB separately. Because `CLONE` and `MERGE` operate on different BLOBs, this simplified notation needs to be extended accordingly. More specifically, we refer to the history of writes H_{global} associated to the BLOB identified by *id* as $H_{id,global}$. Similarly, D_{global} becomes $D_{id,global}$, while $Nodes_{global}$ becomes $Nodes_{id,global}$. Moreover, all functions that access global containers receive *id* additionally as their first parameter, which is used to represent the fact that the functions operate on the global containers of the BLOB identified by *id*.

The `CLONE` primitive is described in Algorithm 11. It simply tests whether the source snapshot v of BLOB *id* was already generated. If this is not the case, failure is returned. Otherwise, it generates a new id *new_id* for the clone and creates its corresponding globally shared containers $H_{new\_id,global}$, $D_{new\_id,global}$ and $Nodes_{new\_id,global}$, which are initialized accordingly. Since the clone shares its data and metadata completely with the original, this initialization step is minimal: a stub entry is inserted in the history of writes to indicate that the clone has a total initial size of t, an empty descriptor map is created (indicating that no update occurred) and finally the root of the segment tree of v is copied into $Nodes_{new\_id,global}$.

`MERGE`, described in Algorithm 12 is very similar to `WRITE`, which is covered in Section 5.4.3. Unlike `WRITE` however, it does not need to write any chunks and generate their corresponding descriptor map, but merge already existing data from a source snapshot *sv* of

Algorithm 11 CLONE

1: **procedure** CLONE($id, v, callback$)
2: **if** ROOT(id, v) = $\emptyset$ **then**
3: **invoke** $callback(failure)$
4: **else**
5: $((v, [x, y]), (v_l, v_r, D)) \leftarrow$ ROOT(id, v)
6: $new\_id \leftarrow$ generate unique id
7: $(t, \_, \_, \_) \leftarrow H_{id,global}[v]$
8: $H_{new\_id,global} \leftarrow \{(1, (t, 0, t, -1))\}$
9: $D_{new\_id,global} \leftarrow \emptyset$
10: $Nodes_{new\_id,global} \leftarrow \{((1, [x, y]), (v_l, v_r, D))\}$
11: **invoke** $callback(new\_id)$
12: **end if**
13: **end procedure**

Algorithm 12 MERGE

1: **procedure** MERGE($sid, sv, soffset, size, did, doffset, callback$)
2: **if** $H_{sid,global}[v] = \emptyset$ **then**
3: **invoke** $callback(-1)$
4: **end if**
5: $(t, \_, \_, \_) \leftarrow H_{sid,global}[v]$
6: **if** $soffset + size > t$ **then**
7: **invoke** $callback(-1)$
8: **end if**
9: $D \leftarrow$ GET_DESCRIPTORS($sid, sv, t, soffset, size$)
10: $i_D \leftarrow$ uniquely generated id
11: $D_{did,global} \leftarrow D_{did,global} \cup (i_D, D)$
12: $(v_a, v_g) \leftarrow$ **invoke remotely** on version manager ASSIGN_WRITE($did, offset, size, i_D$)
13: BUILD_METADATA(did, v_a, v_g, D)
14: **invoke remotely** on version manager COMPLETE(did, v_a)
15: **invoke** $callback(v_a)$
16: **end procedure**

BLOB *sid* to a destination BLOB *did*. This source data is delimited by *soffset* and *size* and is merged into BLOB *did* starting at *doffset*, generating a new snapshot whose version number is returned by the callback. The descriptor map D corresponding to *soffset* and *size* is obtained by inquiring the segment tree of *sid* directly though GET_DESCRIPTORS$(sid, sv, t, soffset, size)$. Once D has been assembled, the process continues in the same way as with writes and appends: D is added to the global descriptor map, a new snapshot version for the merge is requested from the version manager, the new metadata for this new snapshot version is generated by a call to BUILD_METADATA(did, v_a, v_g, D), success is signaled to the version manager and finally the callback is invoked with the new snapshot version as its parameter.

6.4 Example

We consider the same example from Figure 6.2 in order to illustrate the algorithms presented above. We omit illustrating how to read from the segment tree in order to obtain the descriptor map of a subsequence in the BLOB (as this operation is straightforward) and concentrate on the generation of new segment trees. For convenience, Figure 6.2 is repeated in this section as Figure 6.4.

The assumed scenario is the following: after an initial append on to the BLOB (white) which was assigned version 1 (snapshot v_1), three concurrent writers: dark gray, light gray and black start writing to the BLOB. We assume all three have finished writing all their data and have requested each a version number for their update from the version manager. Their assigned version numbers are $2 \ldots 4$, corresponding to the snapshots $v_2 \ldots v_4$. Further, we assume that all three writers were assigned a snapshot version *before* any of them finished writing the metadata, that is, $v_g = 1$ in all cases.

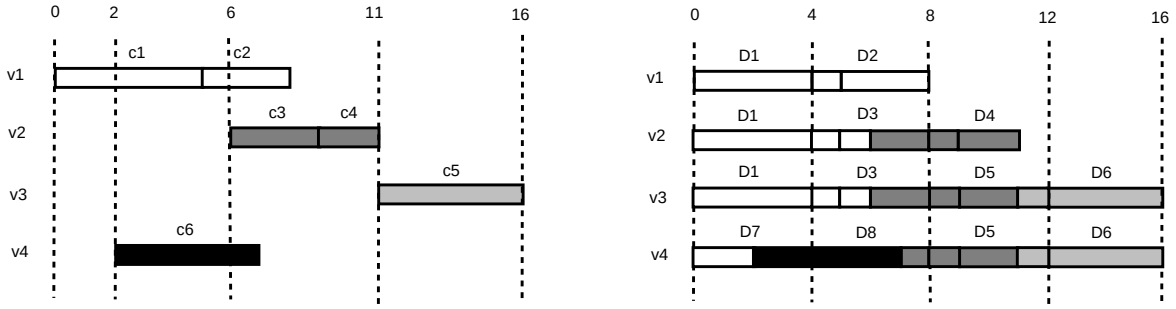
We consider the segment tree generation for black only, leaving the segment tree generation for white, dark gray and light gray as a "homework" to the reader.

In the case of black, $v_a = 4$ and $v_g = 1$, with $D = \{(c6, 0, 5, 2)\}$. The first step in BUILD_METADATA is to extract information about all versions between v_g and v_a from the globally shared history of writes into the local variable W. Next, W is queried for information about black's update: the total size of snapshot: $t_a = 16$, the offset of the update in the BLOB: $o_a = 2$ and the size of the update: $s_a = 5$.

In the first stage, the black leaves are generated. Assuming $lsize = 4$, there are two leaves that overlap with the update of black: the leftmost leaf covers segment $[0, 4]$ while the right most leaf covers segment $[4, 8]$. For each of the leaves, the corresponding descriptor maps must be determined. They are depicted in Figure 6.4(b): D_7 for the leftmost leaf and D_8 for the rightmost leaf. LEAF is responsible to build both D_7 and D_8.

The descriptor map of the leftmost leaf is obtained by calling LEAF$(4, 1, [0, 4], D, W)$. First, we determine what parts of chunks belonging to the black update are covered by the leaf and build their chunk descriptor map D_i. This is the result of INTERSECT$(D, 2, [2, 7] \cap [0, 4], 0)$, which results in $D_i = \{(c6, 0, 2, 2)\}$.

Since the leaf was not fully covered by the update ($Reminder = [0, 2]$), the composition of the remaining part of the leaf is yet to be determined. In order to do that, the history of writes is walked backwards down to v_1 in an attempt to fill *Reminder*. In this case, neither the update of v_3 (which covers $[11, 16]$) nor the update of v_2 (which cov-



(a) History of writes

(b) Descriptor maps of the leaf nodes

Figure 6.4: Four writes/appends (left) and the corresponding composition of the leaves when $lsize = 4$ (right)

ers $[6, 11]$) intersects with $[0, 2]$. However, since the segment tree of v_1 is complete, the composition of *Reminder* can be simply extracted from the descriptor map of the corresponding leaf that covers $[0, 4]$ in v_1, which is D_1. Thus, $D_i = \{(c6, 0, 2, 2)\} \cup$ INTERSECT $(D_1, 0, [0, 2], 0) = \{(c1, 0, 2, 0), (c6, 0, 2, 2)\}$. Assuming the fragmentation threshold k is larger than 1, there is no need to defragment the leaf, thus the result is final and corresponds to D_7.

The descriptor map of the rightmost leaf is obtained in a similar fashion. In this case, the corresponding call to leaf is LEAF$(4, 1, [4, 8], D, W)$. The part of the leaf that is covered the black update is INTERSECT$(D, 2, [2, 7] \cap [4, 8], 4)$, thus $D_i = \{(c6, 2, 3, 0)\}$ and $Reminder = [3, 4]$. Walking backwards through the history of writes reveals v_2 as the most recent snapshot whose segment tree has not been generated yet and whose update fills *Reminder*. Thus, it's descriptor map needs to be consulted in order to extract the corresponding chunk descriptor, leading to $D_i = \{(c6, 2, 3, 0)\} \cup$ INTERSECT $(D_{global}[i_2], 6, [4, 8], 4) = \{(c6, 2, 3, 0), (c3, 1, 1, 3)\}$. Again, there is no need to defragment the leaf, thus D_i is final and equals D_8.

Now that all the leaves have been generated, the second stage consists in building rest of the black segment tree up to the root. Initially,

$$T = \emptyset$$
$$Q = \{((4, [0, 4]), (\_, \_, D_7)), ((4, [4, 8]), (\_, \_, D_8))\}$$

Our algorithm does not make any assumption about which node is extracted first, but for the purpose of this example we assume the first node is extracted. This leads to:

$$T = \{((4, [0, 4]), (\_, \_, D_7))\}$$
$$Q = \{((4, [4, 8]), (\_, \_, D_8))\}$$

Since the first node is the black leftmost leaf, it has a right sibling, which is the black rightmost leaf $(v_4, [4, 8]) \in Q$. Thus, the remaining node in Q is extracted and added to T. Since the version of the sibling is known, the parent of the two black leaves can be determined and is added to Q. At this point,

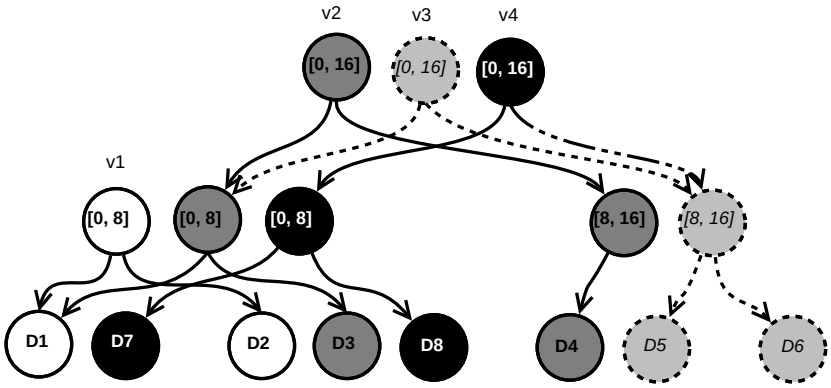


Figure 6.5: Metadata forward references for segment trees: the light-gray segment tree (dotted pattern) has not been generated yet, thus the reference of the black root to its right child (semi-dotted pattern) is a forward reference

$$T = \{((4, [0,4]), (\_, \_, D_7)), ((4, [4,8]), (\_, \_, D_8))\}$$
$$Q = \{((4, [0,8]), (4, 4, \emptyset))\}$$

Since Q is nonempty, a new iteration is performed. The only node in Q is extracted, leaving Q empty. The node is a left child of its parent, thus it has a right sibling which covers $[8, 16]$. This time however its sibling does to belong to Q, thus it it belongs to the segment tree of the most recent lower snapshot version whose update intersects $[8, 16]$. Consulting W reveals this version to be 3, thus the sibling is $(3, [8, 16])$. Note that it does not matter whether this sibling actually exists in order to determine its parent. The reference of the parent to the sibling might be a potential metadata forward reference, as shown in Figure 6.5. The parent is simply added to Q, which leads to:

$$T = \{((4, [0,4]), (\_, \_, D_7)), ((4, [4,8]), (\_, \_, D_8)), ((4, [0,8]), (4, 4, \emptyset))\}$$
$$Q = \{((4, [0,16]), (4, 3, \emptyset))\}$$

Q is again nonempty, thus a new iteration is performed again. The node in Q is extracted, leaving Q empty. Since the node is the root of the segment tree, it is directly added T, leading to the final state:

$$T = \{((4, [0,4]), (\_, \_, D_7)), ((4, [4,8]), (\_, \_, D_8)), ((4, [0,8]), (4, 4, \emptyset)), ((4, [0,16]), (4, 3, \emptyset))\}$$
$$Q = \emptyset$$

Now that Q is empty, the black segment tree generation is complete. T holds the set of black nodes, which is finally added to $Nodes_{global}$.

Chapter 7

Implementation details

Contents

THE algorithms and data structures presented in the previous chapters are based on two high-level abstractions: (1) remote procedure calls that can be executed efficiently in parallel and (2) globally shared containers that support efficient queries on key-value pairs. Furthermore, several important aspects such as fault tolerance, persistency, allocation strategy for new chunks, etc. have been omitted from the presentation in order to simplify understanding.

This chapter discusses the aforementioned details, focusing on the technical side and proposing an event-driven implementation that enables obtaining good performance in practice.

7.1 Event-driven design

As explained in Section 4.2.1, an asynchronous access interface to BLOBs is very important in the context of I/O intensive, distributed applications because it provides better decoupling

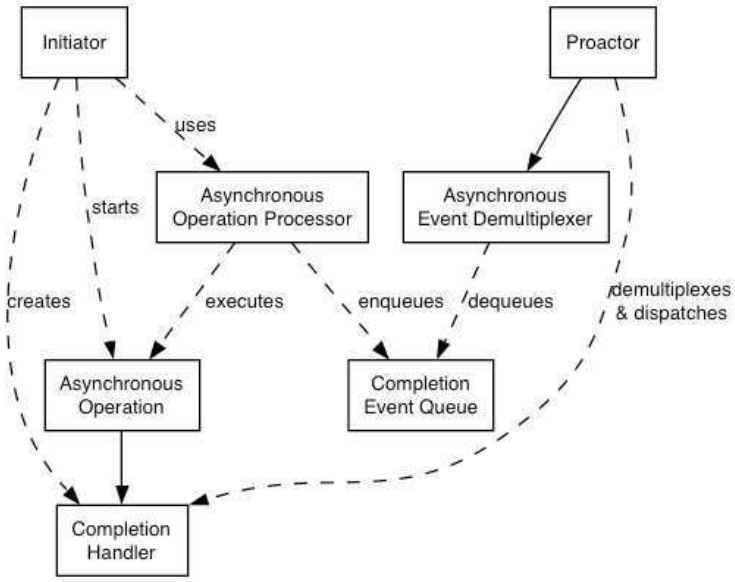


Figure 7.1: Implementing event-driven programming: the proactor pattern

of I/O from computation, which in turn enables scalability to a large number of participants.

Event-driven programming [118, 35] is one of the most robust ways to implement an asynchronous system in practice. Instead of viewing applications as sequential processes that initiate communication among each other, in event-driven programming applications have a reactive behavior: callbacks are executed as a reaction to incoming events.

Event-based systems often bear a strong resemblance to state machines. Each incoming event leads the system into a new state. Understanding a distributed application that is coded as a series of responses to different events is however not easy when there are a large number of interactions, because it is much harder to visualize the execution flow.

Therefore, in order to facilitate a better understanding, an event-driven model was not used explicitly throughout the presentation of the algorithms so far. The use of callbacks was limited to the interactions with the outside, strictly for the purpose of matching with the BLOB access primitive prototypes defined in Section 4.2.1.

However, in the real BlobSeer implementation, event-driven programming is at the core, which enables leveraging asynchrony properly. Our implementation proposal is based on the Proactor pattern [22], illustrated in Figure 7.1. In the Proactor pattern, all asynchronous operation executed by the system are handled by the *asynchronous operation processor*. In order to schedule a new asynchronous operation, the code to be executed together with a *completion handler* (callback) is dispatched to the operation processor. Upon successful completion of an operation, a corresponding completion event is generated by the operation processor and enqueued in the *completion event queue*. In parallel, the *proactor* is responsible to dequeue and demultiplex the events through the *asynchronous event demultiplexer*, which then are dispatched to the corresponding callback as arguments.

The design and execution of callbacks needs careful consideration. Events are independent from each other, bringing the potential to execute their corresponding callbacks in a highly parallel fashion. However, this potential can be achieved only if synchronization

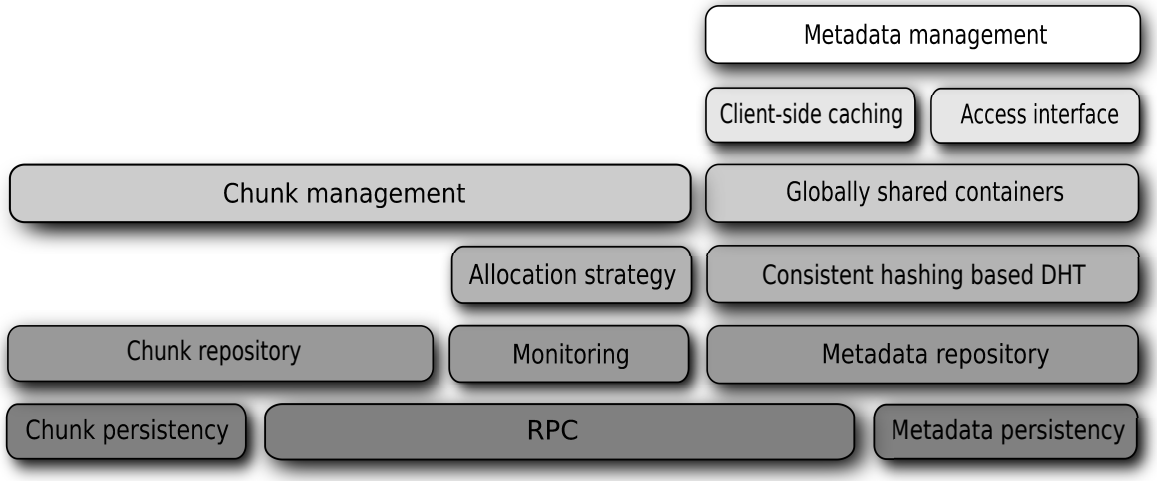


Figure 7.2: Event-based layer design: each layer reacts to the layers below it and generates higher-level events

between callbacks is avoided as much as possible. To decouple callbacks, the BlobSeer implementation adopts several principles borrowed from functional programming: first-order functions, closures and co-routines [148, 8]. These principles bring several benefits in the context of large-scale, distributed applications [146].

The power of event-driven programming comes from the fact that callbacks can act themselves as initiators of asynchronous operations, which enables defining the whole execution flow as reactions to events. The BlobSeer implementation leverages this principle to build event-driven layers, as shown in Figure 7.2. Each layer reacts to events generated by the lower layers and generates itself higher-level events that are reacted upon by the higher layers.

7.1.1 RPC layer

The RPC layer is responsible to mediate communication between all other layers of BlobSeer by means of asynchronous remote procedure calls [3]. Each remote procedure call is initiated by the higher-level layers on the client side, and triggers a corresponding server-side event that encapsulates the parameters of the RPC. Once the server side is done processing the event, it triggers in turn a completion event on the client side. The client side event encapsulates the result and is dispatched to the callback that was associated with the RPC.

All details of communication such as socket management, data transfers and parameter serialization are handled transparently by the RPC layer. We implemented the RPC layer on top of *ASIO* [73], a high-performance asynchronous I/O library that is part of the Boost [1] collection of meta-template libraries. It is based on the event-driven design first introduced in ACE [65].

Since each client needs to communicate with several servers in parallel, a simple solution that opens a new socket for each RPC request does not scale. In order to deal with this issue, we introduced two optimizations:

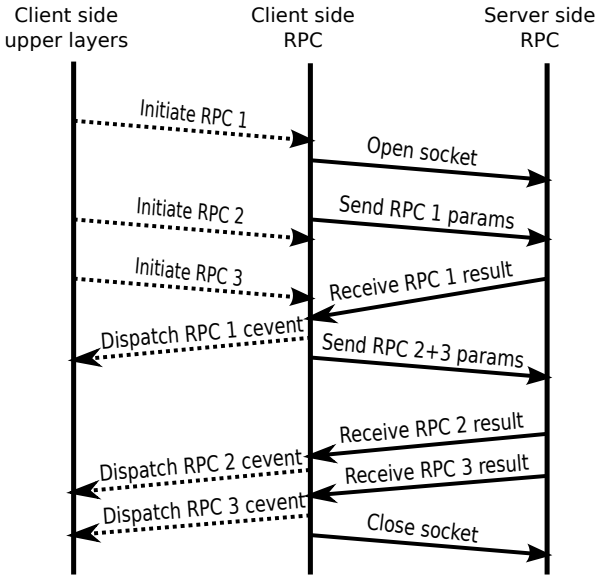


Figure 7.3: RPC layer optimizations: socket reusal, request buffering and aggregation

Socket reusal. Both the client side and the server side of the RPC layer have a limited number of communication channels (TCP sockets) they can use at the same time. These sockets are organized as a socket pool, with the upper limit chosen in such way as to optimize the trade-off between speedup gained by increasing the number of parallel connections, and the slowdown paid for their management. As long as the upper limit is not reached yet, newly initiated RPCs will open and use a new socket. Once the limit is reached, newly initiated RPCs will be buffered and scheduled to reuse the first socket to the same server that becomes available (if such a socket exists) or to open a new socket as soon as possible.

Request aggregation. While waiting for an existing or new socket to become available, it is possible that many RPC requests accumulate for the same destination. In this case, when the socket becomes available, all RPC requests can be aggregated and sent as a single request to the server side. This optimization greatly reduces latency when the number of accumulated RPC requests for the same destination is large.

An example is provided in Figure 7.3. Three RPCs, labeled $1, 2, 3$ are initiated by the client side to the same server. RPC 1 was initiated first and triggered a new socket to be opened. Assuming the limit of the client socket pool has been reached when RPC 2 and 3 are initiated, no more new sockets can be opened and RPC 2 and 3 are buffered. After the reply for RPC 1 arrived, the same socket is reused for RPC 2 and 3. Both RPC requests are aggregated and sent to the server side as a single request. Once the results have been received and the corresponding completion events (*cevents*) dispatched, the socket can be safely closed and its slot in the socket pool freed for other potential requests that wait for opening a new socket.

An important observation about RPC buffering is necessary. Since the parameters of RPCs can grow to large sizes (for example put /get chunk), it is important to use zero-copy

techniques in order to minimize the buffering overhead, both in space and time [92]. Furthermore, choosing the right buffering strategy has also a large impact on performance [21].

7.1.2 Chunk and metadata repositories

The *repository layer* acts as a high-performance remote key-value store, responsible to store and later retrieve chunks (chunk repository) or small metadata objects such as distributed segment tree nodes (metadata repository).

Since from an abstract point of view, chunks are objects themselves, the natural choice might seem to use a universal object repository, such as *Redis* [173] and *MemCached* [45]. While such an approach would have indeed been able to deal with both chunks and metadata objects in an unified fashion, the differences in average size of objects and access pattern motivated us to specialize them into two different layers, as shown in Figure 7.2.

Both repositories form the basic building blocks of data and metadata providers respectively. They rely on the RPC layer to expose a remote get /put object access interface. However, unlike small metadata objects, chunks can grow to large sizes and it is not efficient to fully read a chunk when only a subsequence of it is needed. Therefore, the chunk repository must extend the get primitive to support partial access to chunks.

Again, both repositories leverage the persistency layer to store and retrieve the objects to /from the local disk where the provider is running. We implemented the persistency layer in both cases on top of *BerkeleyDB* [172], a high-performance embedded database. However, a specialized caching mechanism was implemented for each case.

In the case of metadata, the cache needs to hold many small objects entirely, and read /write access to a whole subset of them is common, which makes complex eviction schemes expensive. We found a simple LRU policy to work best in this case. In the case of chunks, the cache needs to hold a smaller amount of larger objects. This setting favors more expensive eviction schemes that use a combination of several criteria, such as the recentness and frequency of use, the size, and the cost of fetching chunks. As pointed out in [121], such eviction schemes have the potential to lead to a sizable improvement in both hit-rate and latency reduction.

7.1.3 Globally shared containers

The *globally shared containers* introduced in Section 5.3 and Section 6.2 require the design of a distributed key-value store that enables efficient exact queries and range queries under concurrency.

We implemented this distributed key-value store in form of a series of layers, as shown in Figure 7.2. At the lowest level is the *distributed hash table (DHT)* layer, responsible to organize the metadata providers (each running a metadata repository) into a DHT. P2P-oriented DHTs [10, 151, 129] that build structured overlays are highly appealing because of their robustness under churn. However, they incur a logarithmic look-up cost to do so, which is prohibitively expensive in our setting. We are interested in almost constant time look-up cost, keeping in mind that metadata providers join and leave at a much smaller scale than in P2P systems. Consistent hashing [69, 70] is a popular approach to achieve

this goal. Our DHT layer implementation is based on consistent hashing, adding a simple replication mechanism that further increases data availability.

The DHT layer by itself can address exact queries out-of-the-box. In order to introduce support for range queries, we built the *globally shared containers layer* on top of the DHT. While this is currently performed as as a series of parallel DHT requests, several more elaborate techniques have been proposed in the literature [36, 31] that could be easily adopted in our context.

Finally, on the client-side, a container access interface enables the client to issue both exact and range queries in an asynchronous fashion. The results of these queries are cached on the client-side as well, and later directly reused if the result is already available in the cache.

7.1.4 Allocation strategy

The *allocation strategy* is the layer upon which the provider manager relies to assign providers to the write /append requests issued by the clients. Its responsibility is to select a data provider for each chunk that needs to be written in the system.

In order to do so, it relies on the *monitoring* layer to collect information about the state of the data providers. Every data provider periodically reports its state to the monitoring layer by means of RPC calls. When the state of a data provider changes, or, if the data provider has not reported in a long time, the monitoring layer generates a corresponding state change event or unavailable event. Such events are captured and processed by the allocation strategy in order to maintain a recent view over the states of the data providers.

Whenever a client queries the provider manager for a list of data providers that need to store chunks, the latter tries to match the client with the "most desirable set of data providers". More specifically, based on information about the client (location, etc.) and the states of the data providers, a score is assigned to each provider that reflects how well that provider can serve the client's write request. All providers are then sorted according to this score and the top providers are assigned to store the chunks, one (distinct if possible) provider for each chunk.

The score calculation is highly customizable and allows easy implementation of different strategies. The default strategy we picked is a load-balancing strategy that favors a provider instead of another if the first provider has stored a smaller number of chunks. In case of equality, the provider with the smallest number of pending write requests (i.e., the smallest amount of times the provider was allocated to clients since it reported its state the last time) is favored. More complex strategies are possible, as illustrated in Chapter 11.

7.2 Fault tolerance

In large-scale distributed systems an important issue is fault tolerance [125, 67, 152]: because of the large number of machines involved, faults invariably occur frequently enough to be encountered in regular use rather than exceptional situations. A desirable property to achieve is then *fault transparency*: based on a self-healing mechanism automatically invoked

when faults occur, the applications can continue their execution without interruption. Additionally, it is also desirable for the system to withstand faults while providing a level of performance close to the case when no faults occur.

In our case, fault transparency refers to two failure scenarios: (1) when a client fails and (2) when a core process (version manager, provider manager, data providers, metadata providers) fails.

The failure model we assume is *permanent crashes* in the first scenario and *permanent or amnesia crashes* in the second. Both types of crashes are described in [34]. A permanent crash is the lack of response starting at a specific moment in time and persisting for an indefinite amount of time. An amnesia crash is the lack of response starting at a specific moment in time as well, but persisting only for a limited amount of time, after which the entity recovers to some stable state in which it was before the crash.

The rationale behind this choice is as follows. Clients are expected to perform read and write operations that take much less time than the down time in case of a crash. Therefore, it is reasonable to assume that a failing client will not recover from a crash fast enough to play an important role in fixing the consequences of the crash. Therefore, it can be safely assumed as "dead forever". Permanent crashes of core processes on the other hand are much more expensive: they involve permanent loss of data or metadata that has to be fixed. Therefore, if the machine where the core process originally ran can be "resurrected" to a stable state, data or metadata initially thought lost can be potentially recovered and reintegrated into the system to fix the problem faster, which makes amnesia crashes worth considering.

7.2.1 Client failures

First, notice that client crashes during read operations can be silently ignored, as they do not alter the state of the system and therefore do not affect data or metadata consistency or otherwise influence other clients.

Write operations on the other hand are not stateless. Any write operation writes data first, and, once this phase is finished, a snapshot version is assigned to the write, then finally the corresponding metadata is generated and stored in a second phase. Thus, if a writer fails during the first phase (i.e. before requesting a snapshot version), the fault can be again silently ignored. The system remains in a consistent state, because nobody else in the system (but the writer) is aware of the new data that was written. These extra data does no harm and can be later garbage-collected in an off-line manner.

Let us now assume that a writer completes the first phase, is assigned a snapshot version, then fails during the second phase. The system is in an inconsistent state because metadata has not fully been built yet. This blocks the generation of the corresponding snapshot and also that of the snapshots corresponding to subsequent write operations. This situation is handled as follows: if metadata takes too long to be generated, a timeout occurs on the version manager, which then delegates metadata generation for that snapshot to a randomly selected metadata provider. Since all required knowledge was already committed into the globally shared containers *before* requesting a snapshot version, any randomly selected metadata provider can take over the metadata generation phase.

7.2.2 Core process failures

To deal with failures of data or metadata providers, we rely on data and metadata replication. Each chunk is replicated on several data providers and metadata is extended to maintain the complete list of providers for each chunk. Similarly, metadata is also replicated on multiple metadata providers by the DHT layer.

Note that the use of versioning and immutable data and metadata greatly simplifies replica management. By isolating readers from writers, replication can be efficiently performed in an asynchronous fashion. There is no need for any complex and costly mechanisms for maintaining replica consistency, as both data chunks and metadata are read-only once written. In order to maintain the replication factor constant, we have chosen an offline approach that relies on monitoring active replicas. As soon as it is detected that a provider goes down, all replicas stored by that provider are marked as unavailable. For each unavailable replica a timer is started. If the replica stays in the unavailable state for too long, a new provider is instructed to fetch a copy of the replica from one of the providers that holds an active copy. Once this is successfully performed, the replica is marked as active again. However, if a provider that was down recovers, it announces all active replicas it holds to the system. If unavailable replicas are among them, they are marked as active again and no new provider is instructed to fetch a copy of it.

Finally, in order to avoid the situation when a failure of any centralized entity (in our case the version manager or provider manager) compromises the whole system, we propose to organize them in small groups that act as replicated state machines running a consensus protocol like [76, 77]. This enables other members of the group to take over in case one of the members fail. At the time of this writing, the replicated state machine has not been implemented in BlobSeer yet.

7.3 Final words

The BlobSeer implementation was written from scratch in C++ using the Boost C++ collection of meta-template libraries [1]. The client-side is implemented as a dynamic library that needs to be linked against the user application. Besides being natively accessible from applications written in C and C++, bindings exist for a wide range of other programming languages: Python, Java and Ruby. More information is available at:

http://blobseer.gforge.inria.fr.

Chapter 8

Synthetic evaluation

Contents

THE BlobSeer implementation described in the previous chapter is evaluated in this chapter through a series of synthetic benchmarks. These benchmarks consist of specific scenarios that focus on each of the design principles presented in Chapter 4. They facilitate the study of their impact on the achieved performance levels under concurrency.

The experiments were performed on the Grid'5000 [68, 26] testbed, an experimental platform that gathers 9 sites in France. We used clusters located in Rennes, Sophia-Antipolis and Orsay. Each experiment was carried out within a single such cluster. The nodes are outfitted with x86_64 CPUs and 4 GB of RAM for the Rennes and Sophia clusters and 2 GB for the Orsay cluster. All nodes are equipped with Gigabit Ethernet, with a measured point-to-point throughput of 117.5 MB/s for TCP sockets with MTU = 1500 B. Latency was measured to be 0.1 ms.

For the purpose of these synthetic experiments, we have chosen to instruct the clients to continuously transfer data without performing any computation, in order to emphasize data access. This corresponds to a *worst-case* theoretical scenario that in practice is not encountered in real applications. It shows the behavior of the I/O infrastructure when it is put under the highest pressure possible.

Each of the next sections focuses on one principle: *data striping*, *distributed metadata management* and *versioning*.

8.1 Data striping

In this section we evaluate the impact of data striping on concurrent access to data. We consider a set of clients that read and write different parts of the same BLOB in parallel, and measure the achieved throughput. The goal of these experiments is to show that under increasing concurrency, data striping successfully distributes the I/O workload among the data providers. This leads to large performance gains over the case when the BLOB is stored in a centralized fashion.

For this purpose, a round-robin allocation strategy was chosen that evenly distributes the chunks among the data providers. Moreover, the chunk size was adjusted to large sizes in order to minimize the size of the segment trees associated to the BLOB and thus minimize the impact of metadata overhead on the results.

The experiments are carried out in two settings: (1) data providers and clients are deployed on different machines; and (2) data providers and clients are co-deployed in pairs on the same physical machine. Both settings are representative of distributed data-intensive applications that separate computation from storage, respectively co-locate computation with storage.

8.1.1 Clients and data providers deployed separately

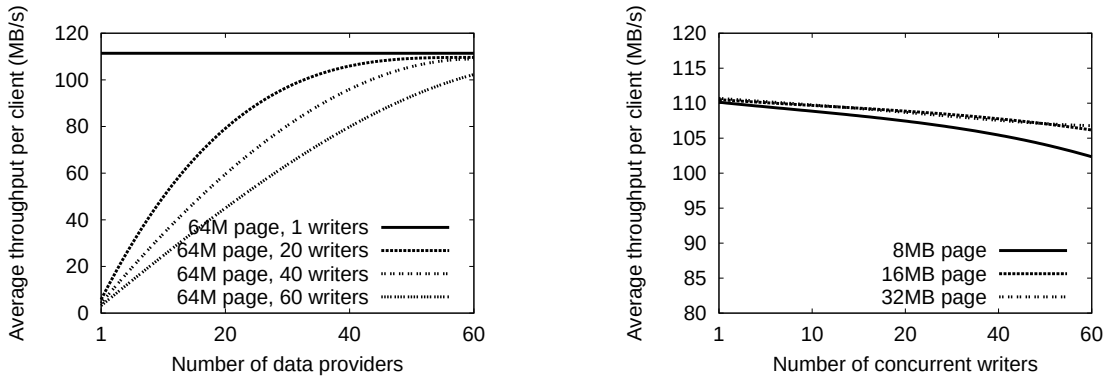
In this series of experiments, the number of concurrent clients is kept constant while the number of available data providers is varied from 1 to 60. The case where a single data provider is available corresponds to the case when the BLOB is stored in a centralized fashion.

The deployment setup is as follows: one version manager, one provider manager and a variable number of data providers. The chunk size is fixed at 64 MB, a size large enough to generate minimal metadata overhead, which can be handled by a single metadata provider and therefore only one is deployed. All processes are deployed on dedicated machines of the Rennes cluster.

The experiment consists in deploying a fixed number of clients on dedicated machines that are then synchronized to start writing each 3 GB of data in chunks of 64 MB at random positions in the same BLOB. Each client writes a single chunk at a time for a total of 48 iterations. The average throughput achieved by the clients is then computed over all their iterations. We start with one data provider and repeat the same experiment by gradually increasing the number of available data providers up to 60.

The experiment described above is performed for 1, 20 and 60 clients, which results in the curves depicted in Figure 8.1(a).

For one single client, using more than one data provider does not make any difference since a single data provider is contacted at the same time. However, when multiple clients concurrently write their output data, the benefits of data striping become visible. Increasing the number of data providers leads to a dramatic increase in achieved throughput per client:



(a) Variable number of available data providers for 1, 20 and 60 concurrent clients

(b) Variable chunk size

Figure 8.1: Impact of data striping on the achieved individual throughput

from a couple of MB/s in the case when the BLOB is stored on a single data provider in a centralized fashion to over 100 MB/s when using 60 data providers.

Throughput gains flatten rapidly when the number of data providers is at least equal to the number of clients. This is explained by the fact that using at least as many data providers as clients enables the provider manager to direct each concurrent write request to distinct data providers. Once the number of providers is higher than the number of clients, no additional gains are observable as the extra providers simply remain idle. When the number of providers is at least equal to the number of clients, the throughput measured under ideal conditions (single client under no concurrency) at 115 MB/s is just by 12% higher than the average throughput reached when 60 clients write the output data concurrently (102 MB/s).

Thus, it can be concluded that data striping effectively enables the system to scale under concurrency. Indeed, the perceived loss of bandwidth under concurrency is minimal from the point of view of clients, as opposed to the case when the BLOB is stored in a centralized fashion, which leads to a huge loss. Although not depicted explicitly, similar results are observed with reads under the same circumstances (i.e., replacing the write operation with a read operation).

As a next step, we evaluate the impact of the chunk size on the achieved average write throughput. This time we fix the number of providers to 60 and deploy a variable number of clients and synchronize them to start writing the same amount of output data in smaller chunks of 32 MB, 16 MB and 8 MB, for a total of 96, 192 and 384 iterations. In this setting, the chunk size is still large enough that the metadata overhead can be considered negligible.

Results are shown in Figure 8.1(b). As can be observed, the overhead of contacting more data providers in parallel and sending the data has a minimal overhead, as the average client bandwidth drops from 110 MB/s (for 32 MB pages) to 102 MB/s (for 8 MB pages).

Thus, it can be concluded that data striping alone enables the system to remain scalable for variable chunk sizes, as long as the total number of involved chunks is small enough so that the metadata management is negligible.

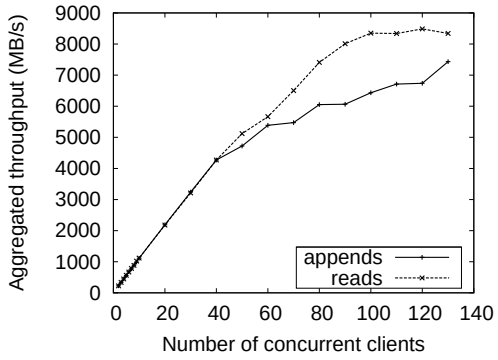


Figure 8.2: Average aggregated read and write throughput of increasing number of concurrent clients

8.1.2 Clients and data providers co-deployed

A similar set of experiments as in the previous section is performed for the case where clients and data providers are co-deployed in pairs, each pair on dedicated physical machine.

Unlike the previous experiment, where the total number of physical machines available in the Rennes cluster limited the number of clients and data providers to 60, this time we can afford to increase the limits further to 130 data providers and clients. We fix and deploy 130 providers and vary the number of co-deployed clients from 1 to 130. The rest of the configuration is the same: one metadata provider, one version manager and one provider manager, each deployed on a dedicated machine.

The chunk size is again fixed at 64 MB. This time however the scenario is slightly different in order to reflect the behavior of data-intensive applications that run in a co-deployed configuration better. More specifically, the clients concurrently append data to an initially empty BLOB. Each client appends a total of 3 GB in 6 iterations, each of which consists in a single append operation of 8 chunks. This access pattern puts even more pressure on the system, because each machine has to execute not only two processes concurrently, but also has to handle more connections simultaneously for each process (compared to the case when only one chunk is transferred at a time).

The same scenario described above was also used to evaluate the read throughput of the system in a co-deployed configuration by replacing the append operation with the read operation. Both reads and appends are depicted in Figure 8.1.2. Unlike the separated deployment scenario, this time we represent the total aggregated throughput achieved by the system, rather than the individual throughput achieved by the clients. This enables a better visualization of our findings at larger scale, where we were able to reach the bandwidth limits of the networking infrastructure.

As can be observed, up to 40 concurrent clients, the curves for reads and appends overlap and scale close to the ideal case (i.e., the throughput of N clients is the throughput of one client multiplied by the number of clients). This is consistent with our findings in the previous section.

However, beyond 40 concurrent clients the gap between reads and appends starts to increase. In the case of reads, a close to ideal scalability is maintained up to 100 machines, when

the total bandwidth of the networking infrastructure is reached (theoretically, 10 GB/s). After the total bandwidth is reached, the aggregated throughput flattens as expected. With a peak aggregated throughput of 8.5 GB/s, our approach has a very low overhead and is close to the real limit (considering the fact that the theoretical limit does not include networking overhead: TCP/IP, frame encapsulation, retransmissions, etc.).

Based on these facts, two important conclusions can be drawn with respect to the benefits of our data striping proposal in the context of highly concurrent read access: (1) it remains highly scalable even when a client and a data provider are co-deployed on the same physical machine and thus share physical resources; and (2) it efficiently leverages the networking infrastructure, pushing it to its limits.

In the case of appends, after 40 clients the scalability is still close to linear but takes a more gentle slope. This effect can be traced back to the persistency layer (described in the previous chapter) of the data providers, that have to cope with increasing write pressure from more concurrent clients. This in turn puts the machine under heavy load, which decreases the responsiveness of the co-deployed client significantly more than in the read scenario.

Nevertheless, with a total aggregated throughput of 7.5 GB/s for 130 concurrent appenders, data striping provides large benefits in a heavily concurrent write scenario, leveraging the total networking infrastructure efficiently as well.

Since point-to-point links are limited to 117 MB/s, storing the BLOB in a centralized fashion cannot achieve a higher aggregated throughput than this under concurrency, which makes it a rather poor choice when compared to data striping, both for intensive read and write scenarios.

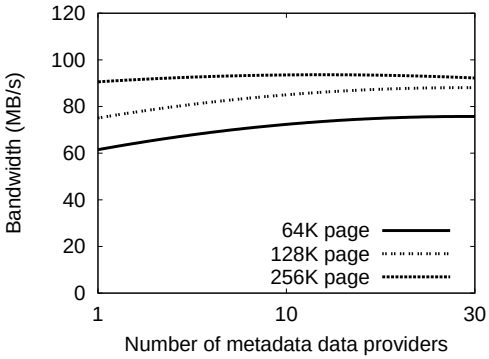
8.2 Distributed metadata management

In the previous set of experiments we have intentionally minimized the metadata management overhead in order to emphasize the impact of data striping on performance. As a next step, we study how metadata decentralization impacts performance: we consider a setting where a large number of clients concurrently read/write a large amount of data in small chunk sizes, which generates a large metadata overhead.

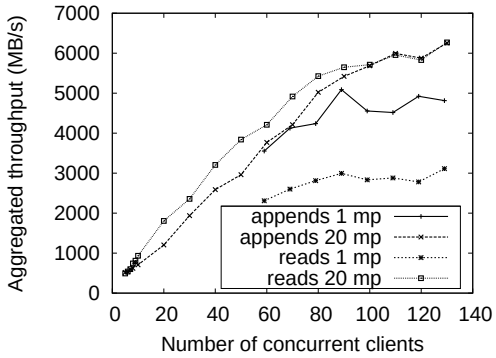
To evaluate the impact of metadata distribution as accurately as possible, we first deploy as many data providers as clients, to avoid potential bottlenecks on the providers. As in the previous section, we carry out the experiments in two settings: (1) data providers and clients are deployed on different machines; and (2) data providers and clients are co-deployed in pairs on the same physical machine.

8.2.1 Clients and data providers deployed separately

In this setting, each process is deployed on a separate physical node: a version manager, a provider manager and a fixed number of 60 data providers. We then launch 60 clients and synchronize them to start writing output data simultaneously. Each client iteratively generates a fixed-sized 64 MB output and writes it to BlobSeer at the same offset (50 iterations). The achieved throughput is averaged for all clients. We vary the number of metadata providers from 1 to 30. We perform this experiment for very small chunk sizes: 64 KB, 128 KB and 256 KB.



(a) Clients and data providers deployed on distinct physical nodes: throughput improvement under concurrency when increasing the number of available metadata providers



(b) Clients and data providers co-deployed: aggregated throughput under concurrency for 1 vs 20 deployed metadata providers.

Figure 8.3: Metadata management efficiency under unfavorable conditions: large number of small chunks

Results in Figure 8.3(a) show that increasing the number of metadata providers results in an improved average bandwidth under heavy concurrency. The improvement is more significant when reducing the chunk size: since the amount of the associated metadata doubles when the page size halves, the I/O pressure on the metadata providers doubles too. We can thus observe that the use of a centralized metadata provider leads to a clear bottleneck (62 MB/s only), whereas using 30 metadata providers improves the write throughput by over 20% (75 MB/s).

Thus, metadata decentralization clearly has an important impact when dealing with a lot of chunks, as significant improvement in throughput is observed from the view point of the clients when increasing the number of metadata providers. Although not depicted explicitly, similar results are observed when the write operation is replaced by the read operation.

8.2.2 Clients and data providers co-deployed

In this setting, data providers and clients are co-deployed, which is representative of distributed applications that co-locate computation with storage. We perform a series of 4 experiments that measure the aggregated throughput achieved when N concurrent clients append 512 MB in chunks of 256 KB (respectively, read 512 MB from disjoint parts of the resulting BLOB).

For each of the experiments, we use the nodes of the Rennes cluster. As many as 130 data providers are deployed on different nodes, with each of the N clients is co-deployed with a data provider on the same node. We consider the case where a single metadata provider is deployed, versus the case where 20 metadata providers deployed on separate nodes, different from the ones where data providers are deployed.

The results are represented in Figure 8.3(b). As can be observed, a distributed metadata management scheme substantially improves access performance, both for readers and appenders. The best improvement occurs is the case of reads, where the total aggregated

throughput has more than doubled. The case of appends has also seen a large improvement: no less than 25%.

8.3 Versioning

In Section 4.1.4 we have argued that versioning is a key principle to enhance concurrent access to data, because multiple versions enable a better isolation, which ultimately decreases the need for synchronization. This is possible because data and metadata is kept immutable, which means reads and writes to/from the same BLOB can be broken into elementary operations that are highly decoupled. Therefore, they do not need to wait for each other, despite the total ordering consistency constraint.

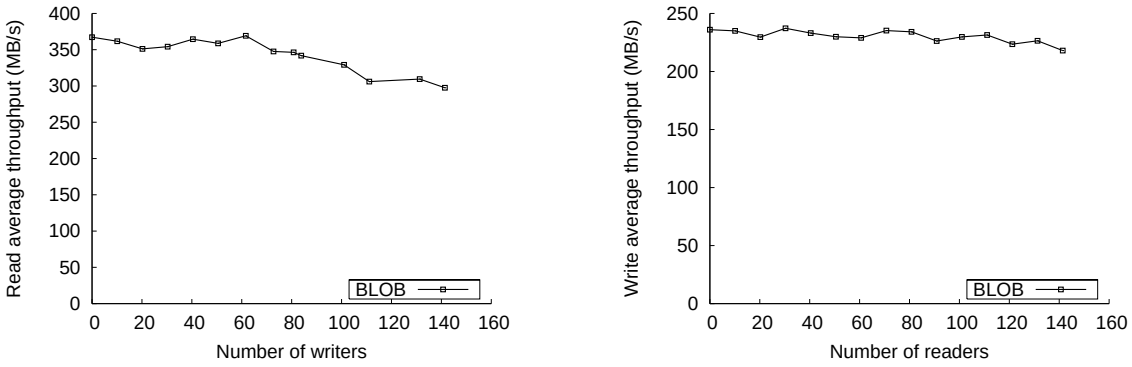
In this section, we perform a series of experiments that put this principle to test. Our goal is to show that even when highly concurrent, mixed read-write workloads are present, BlobSeer can sustain a high throughput and remains scalable in spite of increasing the number of concurrent clients and modifying the read-to-write ratio of the workload.

For this purpose, we perform two experiments that involve a mixed workload where a large number of concurrent clients read and write to the same BLOB. In the first experiment, we fix the number of concurrent readers, and gradually increase the number of concurrent writers. Our goal is to analyze how this impacts the average throughput sustained by the readers. The second experiment is complementary to the first one: we fix the number of concurrent writers and gradually increase the number of concurrent readers in order to analyze how this impacts the average throughput sustained by the writers.

For these two experiments we used the nodes of the Orsay cluster, which totals to 270 nodes. The Ethernet networking infrastructure of the Orsay cluster was insufficient for this series of experiments, as we are using a large number of nodes that are all linked together through a very small number of commodity switches. Fortunately, Orsay is also equipped with a Myrinet networking infrastructure to overcome potential bandwidth limitations. Thus, we decided to use Myrinet for our experiments, as it enables us to emphasize the behavior under concurrency better. Furthermore, it avoids reaching any physical bandwidth limitations that could interfere with the conclusions that can be drawn about scalability.

In order to be able to deploy as many concurrent clients as possible, we opted for a co-deployed scenario, where the clients and data providers run on the same physical node. Thus, for each of the two experiments, BlobSeer was deployed on the 270 nodes in the following configuration: one version manager, one provider manager and 20 metadata providers. The rest of the nodes were used to co-deploy data providers and clients.

In the first experiment, 100 readers are deployed and synchronized to concurrently start reading 10 chunks of 64MB from the same BLOB. Each reader accesses a different region of the BLOB. At the same time, an increasing number of writers is deployed ranging from 0 (only readers) to 140. Each writer generates and writes 10 chunks of 64MB. Results are shown in Figure 8.4(a). As can be observed, the average sustained throughput of the readers drops from 370 MB/s to 300 MB/s when the number of concurrent writers increases from 0 to 140. Thus, we can conclude that we have a drop of only 20% in a situation where more than twice as many clients access the same BLOB concurrently and generate a mixed read-write workload. These results are even more impressive considering that we use a scenario where a small amount of data is processed that can be cached on the data providers. No



(a) Impact of concurrent writes on concurrent reads from the same file

(b) Impact of concurrent reads on concurrent writes to the same file

Figure 8.4: Benefits of versioning under mixed, highly concurrent read and write workloads

disk activity that could limit throughput is involved, which makes the system even more sensitive to scalability issues.

Similar results are obtained in the second experiment, where the readers and the writers reverse roles. This time, we fix the number of concurrent writers to 100 and increase the number of concurrent readers from 0 to 140. The access pattern for readers and writers remains the same as in the previous experiment. Results are shown in Figure 8.4(b). Write operations are slower as there is a higher co-deployment overhead, since data has to be asynchronously committed to disk. For this reason, the scalability in this case is even better: introducing more than double concurrent readers causes the average write throughput to drop by only 10%.

8.4 Conclusions

We performed extensive synthetic benchmarks using BlobSeer that emphasize the impact of each of the proposed design principles (data striping, distributed metadata management and versioning) on the sustained throughput under concurrent access to the same BLOB.

Data striping shows best results when at least as many data providers are deployed as clients, because each client can interact with a different provider. In this case, average throughput under concurrency is at most 12% lower than the case where no concurrency is present. Aggregated throughput measurements show that BlobSeer reached the physical limits of the networking infrastructure, demonstrating that it can leverage it efficiently.

The distributed metadata management scheme shows best results when a large number of chunks is involved in the data transfers. In this case, speedup against centralized metadata management under concurrency is at least 20%.

Finally, our versioning proposal shows high scalability under mixed read-write workloads. In this context, the drop in average throughput when doubling the number of concurrent clients is not more than 20% in the worst case and close to 10% in the best case.

Part III

Applications of the BlobSeer approach

Chapter 9

High performance storage for MapReduce applications

Contents

As the rate, scale and variety of data increases in complexity, new data-intensive paradigms are required in order to achieve high performance level at large scale. Several paradigms such as MapReduce [38] and Dryad [66] have been proposed to address this need. MapReduce has been hailed as a revolutionary new platform for large-scale, massively parallel data access [116]. After being strongly promoted by

Google, MapReduce has also been implemented by the open source community through the Hadoop [169, 164] project, maintained by the Apache Foundation and supported by Yahoo! and even by Google itself. This model is currently getting more and more popular as a solution for rapid implementation of distributed data-intensive applications. A key component of MapReduce upon which the performance of the whole model depends is the storage backend, as it has to deal efficiently with the massively parallel data access. Thus, evaluating BlobSeer as a storage backend for MapReduce applications is a highly relevant context that enables us to demonstrate its benefits in real life applications. In this chapter we show how we successfully achieved this objective. The work presented here was published in [112].

9.1 BlobSeer as a storage backend for MapReduce

9.1.1 MapReduce

MapReduce proposes to exploit parallelism at data level *explicitly*, by forcing the user to design the application according to a predefined model [75], inspired by the *map* and *reduce* primitives commonly used in functional programming, although their purpose in MapReduce is not the same as their original forms.

A problem must be expressed in MapReduce using two operators: "map" and "reduce". The map operator defines a transformation that is applied by the framework in parallel to the input data in order to obtain a set of intermediate key-value pairs as output. The reduce operator defines an aggregation method for values that correspond to the same key in the intermediate set of key-value pairs. The final result of the MapReduce application is the aggregated set of key-value pairs after applying the reduce operator for all keys.

This model is generic enough to cover a large class of applications, with supporters claiming the extreme that it will render relational database management systems obsolete. At least one enterprise, Facebook, has implemented a large data warehouse system using MapReduce technology rather than a DBMS [159].

While some work points out the limitations of MapReduce [117, 153] compared to DBMS approaches, it still acknowledges the potential of MapReduce to complement DBMSes. Ongoing work even exists [114] to build a new query language, PigLatin, designed to fit in the sweet-spot between the declarative style of SQL [17], and the low-level, procedural style of MapReduce. The intent is to compile PigLatin into physical plans that are executed over MapReduce, greatly simplifying the development and execution of data analysis tasks and bringing the MapReduce framework closer to the DBMS concept.

The main reason for the popularity of MapReduce is its high potential for scalability: once the application is cast into the framework, all details of distributing the application are automatically handled: splitting the workload among the compute nodes, synchronization, data access, fault tolerance, assembly of the final result, etc. This is very appealing in large-scale, distributed environments, as it greatly simplifies application design and development. The burden for enabling this feature however falls on the framework itself: all aforementioned aspects have to be addressed efficiently with minimal user intervention.

To enable massively parallel data access over a large number of nodes, the framework relies on a core component: the storage backend. Since MapReduce applications are generally

data-intensive, the storage backend plays a crucial role in the overall scalability and performance of the whole framework. It must meet a series of specific requirements which are not part of the design specifications of traditional distributed file systems employed in the HPC communities: these file systems typically aim at conforming to well-established standards such as POSIX and MPI-IO.

9.1.2 Requirements for a MapReduce storage backend

MapReduce applications typically crunch ever-growing datasets of billions of small records. Storing billions of KB-sized records in separate tiny files is both unfeasible and hard to handle, even if the storage backend supports it. For this reason, the datasets are usually packed together in *huge files* whose size reaches several hundreds of GB.

The key strength of the MapReduce model is its inherently high parallelization of the computation, which enables processing Petabytes of data in a couple of hours on large clusters consisting of several thousands of nodes. This has several consequences for the storage backend. Firstly, since data is stored in huge files, the computation will have to process small parts of these huge files concurrently. Thus, the storage backend is expected to provide efficient *fine-grain access* to the files. Secondly, the storage backend must be able to sustain a *high throughput* in spite of *heavy access concurrency* to the same file, as thousands of clients access data simultaneously.

Dealing with huge amounts of data is difficult in terms of manageability. Simple mistakes that may lead to loss of data can have disastrous consequences since gathering such amounts of data requires considerable effort. In this context, *versioning* becomes an important feature that needs to be supported by the storage backend. Not only does it enable rolling back undesired changes, but it also enables profiling a MapReduce execution by analysing changes from one version to another, which may later prove useful in improving the application. While versioning is certainly an interesting feature, it should have a minimal impact both on performance and on storage space overhead.

Finally, another important requirement for the storage backend is its ability to expose an interface that enables the application to be *data-location aware*. This allows the scheduler to use this information to place computation tasks close to the data. This reduces network traffic, contributing to a better global data throughput.

9.1.3 Integrating BlobSeer with Hadoop MapReduce

To address the requirements presented in the previous section, specialized filesystems have been designed, such as GoogleFS [53], used as a backend for the proprietary Google MapReduce implementation, and HDFS [144], the default storage backend of the Hadoop [169] framework. We have briefly presented both filesystems in Section 3.4.3.

Since the Hadoop framework is open-source and was adopted by large players in the industry such as Yahoo! and Facebook, it stands as a good candidate for exploring the role of BlobSeer as a storage backend for MapReduce. The challenge thus is to extend BlobSeer to the point where it is possible to substitute HDFS with it. Essentially, this means to enable BLOBs to be used as files in Hadoop MapReduce.

The Hadoop MapReduce framework provides a specific Java API by which the underlying storage service is accessed. This API exposes the basic operations of a file system: create directory/file, list directory, open/close/read/write file, etc. However, it does not aim to be POSIX compatible and introduces a series of advanced features, such as atomic appends to the same file.

In order to enable BLOBs to be used as files, this API had to be implemented in a dedicated backend on top of BlobSeer. We call this backend the *BlobSeer File System* (BSFS). To enable a fair comparison between BSFS and HDFS, we addressed several performance-oriented issues highlighted in [170]. They are briefly discussed below.

File system namespace. The Hadoop framework expects a classical hierarchical directory structure, whereas BlobSeer provides a flat structure for BLOBs. For this purpose, we had to design and implement a specialized *namespace manager*, which is responsible for maintaining a file system namespace, and for mapping files to BLOBs. For the sake of simplicity, this entity is centralized. Careful consideration was given to minimizing the interaction with this namespace manager, in order to fully benefit from the decentralized metadata management scheme of BlobSeer. Our implementation of Hadoop's file system API only interacts with it for operations like file opening and file/directory creation/deletion/renaming. Access to the actual data is performed by a direct interaction with BlobSeer through read/write/append operations on the associated BLOB, which fully benefit from BlobSeer's efficient support for concurrency.

Data prefetching. Hadoop manipulates data sequentially in small chunks of a few KB (usually, 4 KB) at a time. To optimize throughput, HDFS implements a caching mechanism that prefetches data for reads, and delays committing data for writes. Thereby, physical reads and writes are performed with data sizes large enough to compensate for network traffic overhead. We implemented a similar caching mechanism in BSFS. It prefetches a whole chunk when the requested data is not already cached, and delays committing writes until a whole chunk has been filled in the cache.

Affinity scheduling: exposing data distribution. In a typical Hadoop deployment, the same physical nodes act both as storage elements and as computation workers. Therefore, the Hadoop scheduler strives at placing the computation as close as possible to the data: this has a major impact on the global data throughput, given the huge volume of data being processed. To enable this scheduling policy, Hadoop's file system API exposes a call that allows Hadoop to learn how the requested data is split into chunks, and where those chunks are stored. We address this point by extending BlobSeer with a new primitive. Given a specified BLOB id, version, offset and size, it returns the list of chunks that make up the requested range, and the addresses of the physical nodes that store those chunks. Then, we simply map Hadoop's corresponding file system call to this primitive provided by BlobSeer.

9.2 Experimental setup

9.2.1 Platform description

To evaluate the benefits of using BlobSeer as the storage backend for MapReduce applications we used Yahoo!'s release of Hadoop v.0.20.0 (which is essentially the main release of Hadoop with some minor patches designed to enable Hadoop to run on the Yahoo! production clusters). We chose this release because it is freely available and enables us to experiment with a framework that is both stable and used in production on Yahoo!'s clusters.

We performed our experiments on the Grid'5000 [68], using the clusters located in Sophia-Antipolis, Orsay and Lille. Each experiment was carried out within a single such cluster. The nodes are outfitted with x86_64 CPUs and 4 GB of RAM for the Rennes and Sophia clusters and 2 GB for the Orsay cluster. Intracluster bandwidth is 1 Gbit/s (measured: 117.5 MB/s for TCP sockets with MTU = 1500 B), intracluster latency is 0.1 ms. A significant effort was invested in preparing the experimental setup, by defining automated deployment processes for the Hadoop framework when using respectively BlobSeer and HDFS as the storage backend. We had to overcome nontrivial node management and configuration issues to reach this objective.

9.2.2 Overview of the experiments

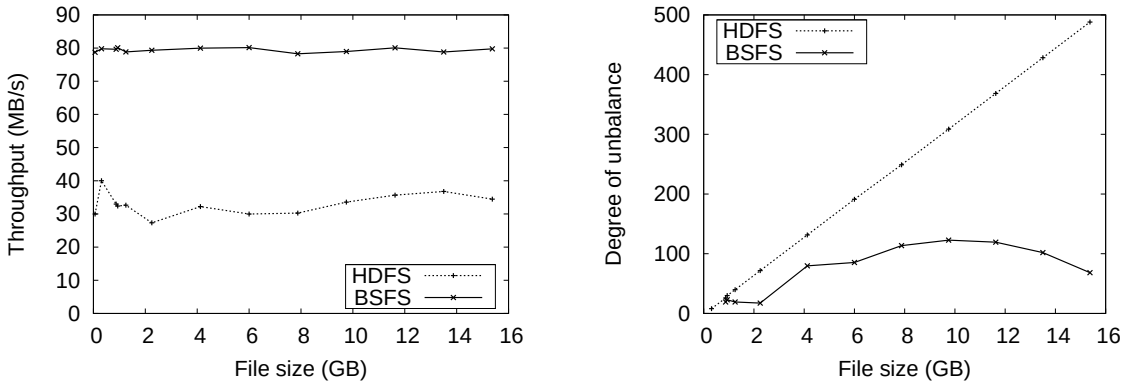
In a first phase, we have implemented a set of microbenchmarks that write, read and append data to files through Hadoop's file system API and have measured the achieved throughput as more and more concurrent clients access the file system. This synthetic setup has enabled us to control the access pattern to the file system and focus on different scenarios that exhibit particular access patterns. We can thus directly compare the respective behavior of BSFS and HDFS in these particular synthetic scenarios.

In a second phase, our goal was to get a feeling of the impact of BlobSeer at the application level. To this end, we ran two standard MapReduce applications from the Hadoop release, both with BSFS and with HDFS. We have evaluated the impact of using BSFS instead of HDFS on the total job execution time as the number of available MapReduce workers progressively increases. Note that Hadoop MapReduce applications run out-of-the-box in an environment where Hadoop uses BlobSeer as a storage backend, just like in the original, unmodified environment of Hadoop. This was made possible thanks to the Java file system interface we provided with BSFS on top of BlobSeer.

9.3 Microbenchmarks

We have first defined several scenarios aiming at evaluating the throughput achieved by BSFS and HDFS when the distributed file system is accessed by a single client or by multiple, concurrent clients, according to several specific access patterns. In this section, we report on the results focused on the following patterns, often exhibited by MapReduce applications:

- a single process writing a huge distributed file;
- concurrent readers reading different parts of the same huge file;



(a) Performance of HDFS and BSFS when a single client writes to a single file

(b) Load-balancing evaluation

Figure 9.1: Single writer results

- concurrent writers appending data to the same huge file.

The aim of these experiments is to evaluate which benefits can be expected when using a concurrency-optimized storage service such as BlobSeer for highly-parallel MapReduce applications generating such access patterns. The relevance of these patterns is discussed in the following subsections, for each scenario.

In each scenario, we first measure the throughput achieved when a single client performs a set of operations on the file system. Then, we gradually increase the number of clients performing the same operation concurrently and measure the average throughput per client. For any fixed number N of concurrent clients, the experiment consists of two phases: we deploy HDFS (respectively BSFS) on a given setup, then we run the benchmarking scenario.

In the deployment phase, HDFS (respectively BSFS) is deployed on 270 machines from the same cluster of Grid'5000. For HDFS, we deploy one namenode on a dedicated machine; the remaining nodes are used for the datanodes (one datanode per machine). For the BSFS deployment we use the same nodes in the following configuration: one version manager, one provider manager, one node for the namespace manager, 20 metadata providers and the remaining nodes are used as data providers. Each entity is deployed on a a separate, dedicated machine.

For the benchmarking phase, a subset of N machines is selected from the set of machines where datanodes/providers are running. The clients are then launched simultaneously on this subset of machines, individual throughput is collected and is then averaged. These steps are repeated 5 times for better accuracy (which is enough, as the corresponding standard deviation proved to be low). Measurements in this second phase of the experiment are performed for both HDFS and BSFS.

9.3.1 Single writer, single file

We first measure the performance of HDFS/BSFS when a single client writes a file whose size gradually increases. This test consists in sequentially writing a unique file of $N \times 64$ MB,

in chunks of 64 MB (N goes from 1 to 246), which corresponds to the default chunk size of HDFS. The goal of this experiment is to compare the chunk allocation strategies of HDFS and BSFS used in distributing the data across datanodes (respectively data providers). The policy used by HDFS consists in writing *locally* whenever a write is initiated on a datanode. To enable a fair comparison, we chose to always deploy clients on nodes where no datanode has previously been deployed. This way, we make sure that HDFS will distribute the data among the datanodes, instead of locally storing the whole file. BlobSeer's default strategy consists in allocating the corresponding chunks on remote providers in a round-robin fashion.

We measure the write throughput for both HDFS and BSFS: the results can be seen on Figure 9.1(a). BSFS achieves a significantly higher throughput than HDFS, which is a result of the balanced, round-robin chunk distribution strategy used by BlobSeer. A high throughput is sustained by BSFS even when the file size increases (up to 16 GB). To evaluate the load balancing in both HDFS and BSFS, we chose to compute the *Manhattan distance* to an ideally balanced system where all data providers/datanodes store the same number of chunks. To calculate this distance, we represent the data layout in each case by a vector whose size is equal to the number of data providers/datanodes; the elements of the vector represent the number of chunks stored by each provider/datanode. We compute 3 such vectors: one for HDFS, one for BSFS and one for a perfectly balanced system (where all elements have the same value: the total number of chunks divided by the total number of storage nodes). We then compute the distance between the "ideal" vector and the HDFS (respectively BSFS) vector.

As shown on Figure 9.1(b), as the file size (and thus, the number of chunks) increases, both BSFS and HDFS become unbalanced. However, BSFS remains much closer to a perfectly balanced system, and it manages to distribute the chunks almost evenly to the providers, even in the case of a large file. As far as we can tell, this can be explained by the fact that the chunk allocation policy in HDFS mainly takes into account data locality and does not aim at perfectly balancing the data distribution. A global load-balancing of the system is done for MapReduce applications when the tasks are assigned to nodes. During this experiment, we could notice that in HDFS there are datanodes that do not store any chunk, which explains the increasing curve shown in figure 9.1(b). As we will see in the next experiments, a balanced data distribution has a significant impact on the overall data access performance.

9.3.2 Concurrent reads, shared file

In this scenario, for each given number N of clients varying from 1 to 250, we executed an experiment in two steps. First, we performed a boot-up phase, where a single client writes a file of $N \times 64$ MB, right after the deployment of HDFS/BSFS. Second, N clients read parts from the file concurrently; each client reads a different 64 MB chunk sequentially, using finer-grain blocks of 4 KB. This pattern where multiple readers request data in chunks of 4 KB is very common in the "map" phase of a Hadoop MapReduce application, where the mappers read the input file in order to parse the (key, value) pairs.

For this scenario, we ran two experiments in which we varied the data layout for HDFS. The first experiment corresponds to the case where the file read by all clients is entirely stored by a single datanode This corresponds to the case where the file has previously been entirely written by a client colocated with a datanode (as explained in the previous scenario). Thus, all clients subsequently read the data stored by one node, which will lead to a very poor

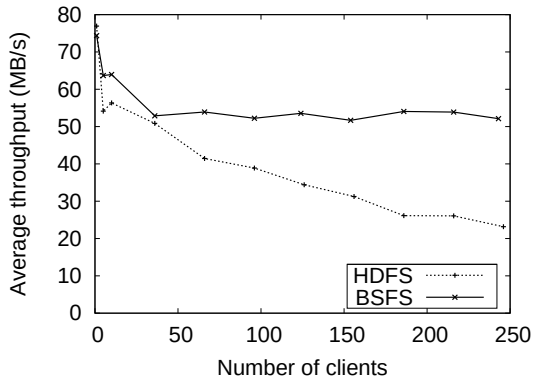


Figure 9.2: Performance of HDFS and BSFS when concurrent clients read from a single file

performance of HDFS. We do not report on these results here. In order to achieve a more fair comparison where the file is distributed on multiple nodes both in HDFS and in BSFS, we chose to execute a second experiment. Here, the boot-up phase is performed on a dedicated node (no datanode is deployed on that node). By doing so, HDFS will spread the file in a more balanced way on multiple remote datanodes and the reads will be performed remotely for both BSFS and HDFS. This scenario also offers an accurate simulation of the first phase of a MapReduce application, when the mappers are assigned to nodes. The HDFS job scheduler tries to assign each map task to the node that stores the chunk the task will process; these tasks are called *local maps*. The scheduler also tries to achieve a global load-balancing of the system, such that not all the assignments will be local. The tasks running on a different node than the one storing its input data, are called *remote maps*: they will read the data remotely.

The results obtained in the second experiment are presented on Figure 9.2. BSFS performs significantly better than HDFS, and moreover, it is able to deliver the same throughput even when the number of clients increases. This is a direct consequence of how balanced is the chunk distribution for that file. The superior load balancing strategy used by BlobSeer when writing the file has a positive impact on the performance of concurrent reads, whereas HDFS suffers from the poor distribution of the file chunks.

9.3.3 Concurrent appends, shared file

We now focus on another scenario, where multiple, concurrent clients append data to the same file. The Hadoop API explicitly offers support for atomic appends in this context, as concurrent appends are useful not only in order to optimize the MapReduce framework itself (for example, appending the results of parallel reducers in the same output file), but also for data gathering applications that build the initial input of MapReduce jobs (for example, web crawlers).

As BlobSeer provides support for concurrent appends by design, we have implemented the append operation in BSFS and evaluated the aggregated throughput as the number of clients varies from 1 to 250. Despite being part of the Hadoop API specification, append support was not available in HDFS at the time of this writing. Therefore, we could not perform the same experiment for HDFS.

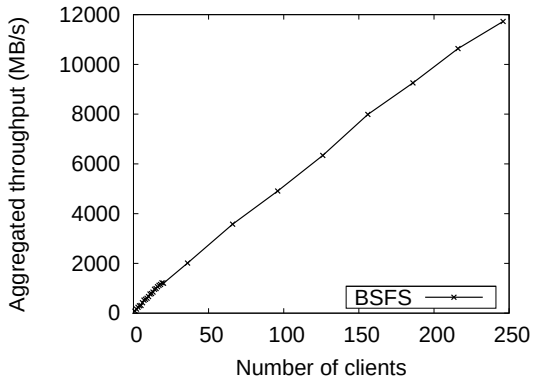
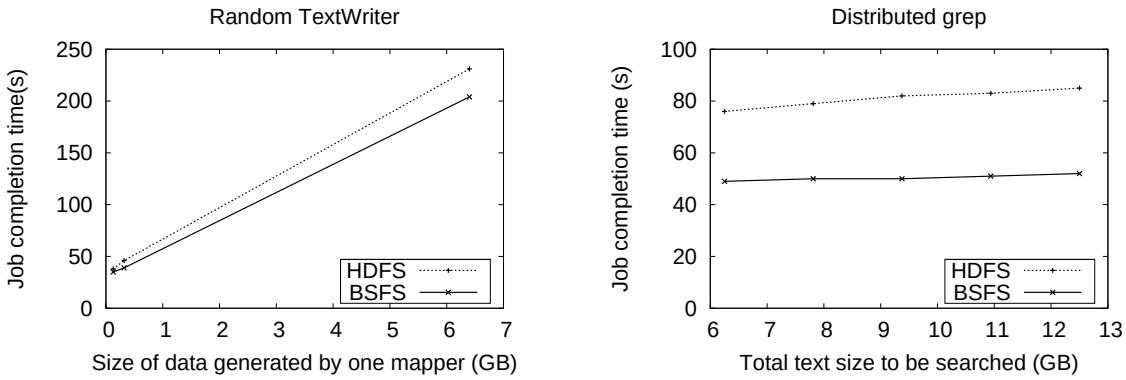


Figure 9.3: Performance of BSFS when concurrent clients append to the same file



(a) RandomTextWrite: Job completion time for a total of 6.4 GB of output data when increasing the data size generated by each mapper

(b) Distributed grep: Job completion time when increasing the size of the input text to be searched

Figure 9.4: Benefits of using BSFS instead of HDFS as a storage backend in Hadoop: impact on the performance of MapReduce applications

Figure 9.3 illustrates the aggregated throughput obtained when multiple clients concurrently append data to the same BSFS file. These good results can be obtained thanks to BlobSeer, which is optimized for concurrent appends.

Note that these results also give an idea about the performance of concurrent writes to the same file. In BlobSeer, the append operation is implemented as a special case of the write operation where the write offset is implicitly equal to the current file size: the underlying algorithms are actually identical. The same experiment performed with writes instead of appends, leads to very similar results.

9.4 Higher-level experiments with MapReduce applications

In order to evaluate how well BSFS and HDFS perform in the role of storage backends for real MapReduce applications, we selected a set of standard MapReduce applications that are

part of Yahoo!'s Hadoop release.

9.4.1 RandomTextWriter

The first application, *RandomTextWriter*, is representative of a distributed job consisting in a large number of tasks each of which needs to write a large amount of output data (with no interaction among the tasks). The application launches a fixed number of mappers, each of which generates a huge sequence of random sentences formed from a list of predefined words. The reduce phase is missing altogether: the output of each of the mappers is stored as a separate file in the file system. The access pattern generated by this application corresponds to concurrent, massively parallel writes, each of them writing to a different file.

To compare the performance of BSFS vs.HDFS in such a scenario, we co-deploy a Hadoop tasktracker with a datanode in the case of HDFS (with a data provider in the case of BSFS) on the same physical machine, for a total of 50 machines. The other entities for Hadoop, HDFS (namenode, jobtracker) and for BSFS (version manager, provider manager, namespace manager) are deployed on separate dedicated nodes. For BlobSeer, 10 metadata providers are deployed on dedicated machines as well.

We fix the total output size of the job at 6.4 GB worth of generated text and vary the size generated by each mapper from 128 MB (corresponding to 50 parallel mappers) to 6.4 GB (corresponding to a single mapper), and measure the job completion time in each case.

Results obtained are displayed on Figure 9.4(a). Observe that the relative gain of BSFS over HDFS ranges from 7 % for 50 parallel mappers to 11 % for a single mapper. The case of a single mapper clearly favors BSFS and is consistent with our findings for the synthetic benchmark in which we explained the respective behavior of BSFS and HDFS when a single process writes a huge file. The relative difference is smaller than in the case of the synthetic benchmark because here the total job execution time includes some computation time (generation of random text). This computation time is the same for both HDFS and BSFS and takes a significant part of the total execution time.

9.4.2 Distributed grep

The second application we consider is *distributed grep*. It is representative of a distributed job where huge input data needs to be processed in order to obtain some statistics. The application scans a huge text input file for occurrences of a particular expression and counts the number of lines where the expression occurs. Mappers simply output the value of these counters, then the reducers sum up the all the outputs of the mappers to obtain the final result. The access pattern generated by this application corresponds to concurrent reads from the same shared file.

In this scenario we co-deploy a tasktracker with a HDFS datanode (with a BlobSeer data provider, respectively), on a total of 150 nodes. We deploy all centralized entities (version manager, provider manager, namespace manager, namenode, etc.) on dedicated nodes. Also, 20 metadata providers are deployed on dedicated nodes for BlobSeer.

We first write a huge input file to HDFS and BSFS respectively. In the case of HDFS, the file is written from a node that is not colocated with a datanode, in order to avoid the scenario where HDFS writes all chunks locally. This gives HDFS the chance to perform some

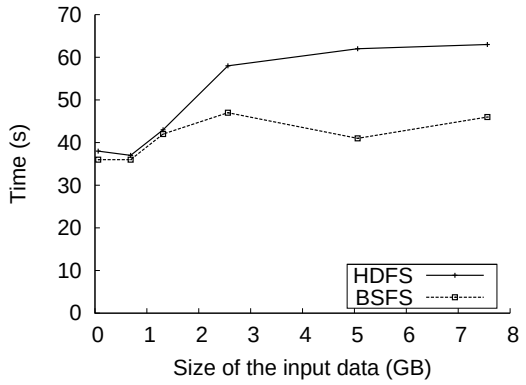


Figure 9.5: Sort: job completion time

load-balancing of its chunks. Then we run the distributed grep MapReduce application and measure the job completion time. We vary the size of the input file from 6.4 GB to 12.8 GB in increments of 1.6 GB. Since the default HDFS chunk is 64 MB large and usually Hadoop assigns a single mapper to process each chunk, this roughly corresponds to varying the number of concurrent mappers from 100 to 200.

The obtained results are presented on Figure 9.4(b). As can be observed, BSFS outperforms HDFS by 35 % for 6.4 GB and the gap steadily increases with the text size. This behavior is consistent with the results obtained for the synthetic benchmark where concurrent processes read from the same file. Again, the relative difference is smaller than in the synthetic benchmark because the job completion time accounts for both the computation time and the I/O transfer time. Note however the high impact of I/O in such applications that scan through the data for specific patterns: the benefits of supporting efficient concurrent reads from the same file at the level of the underlying distributed file system are definitely significant.

9.4.3 Sort

Finally, we evaluate *sort*, a standard MapReduce application, that sorts key-value pairs. The key is represented by the first 10 bytes from each record, while the value is the remaining 100 bytes. This application is read-intensive in the *map* phase and it generates a write-intensive workload in the *reduce* phase. The access patterns exhibited by this application are thus *concurrent reads from the same file* and *concurrent writes to different files*.

A full deployment of HDFS/BSFS was performed on all 270 available nodes followed by a deployment of the entities belonging to the Hadoop framework: the jobtracker, deployed on a dedicated node, and the tasktrackers, co-deployed with the datanodes/providers. The input file to be sorted by the application is stored in 64 MB chunks spread across the datanodes/providers. The Hadoop jobtracker assigns a mapper to process each chunk of the input file. The same input data was stored in multiple chunk configurations in order to be able to vary the number of mappers from 1 to 120. This corresponds to an input file whose size varies from 64 MB to 8 GB. For each of these input files, we measured the job completion time when HDFS and BSFS are respectively used as storage backends.

Figure 9.5 displays the time needed by the application to complete, when increasing the size of the input file. When using BSFS as a storage backend, the Hadoop framework manages to finish the job faster than when using HDFS. These results are consistent with the ones delivered by the microbenchmarks. However, the impact of the average throughput when accessing a file in the file system is less visible in these results, as the job completion time includes not only file access time, but also the computation time and the I/O transfer time.

9.5 Conclusions

In this chapter we presented BlobSeer-based File System (BSFS), a storage layer for Hadoop MapReduce that builds on BlobSeer to provide high performance and scalability for data-intensive applications. We demonstrated that it is possible to enhance Hadoop MapReduce by replacing the default storage layer, Hadoop Distributed File System (HDFS), with BSFS. Thank to this new BlobSeer-based File System (BSFS) layer, the sustained throughput of Hadoop is significantly improved in scenarios that exhibit highly concurrent accesses to shared files. We demonstrated this claim through extensive experiments, both using synthetic benchmarks and real MapReduce applications. The results obtained in the synthetic benchmarks show not only large throughput improvements under concurrency, but also superior scalability and load balancing. These theoretical benefits were put to test by running real-life MapReduce applications that cover all possible access pattern combinations: read-intensive, write-intensive and mixed. In all three cases, improvement over HDFS ranges from 11% to 30%.

Chapter 10

Efficient VM Image Deployment and Snapshotting in Clouds

Contents

In this chapter we leverage the object-versioning capabilities of BlobSeer to address two important challenges that arise in the context of IaaS cloud computing (presented in Section 2.3): (1) efficient deployment of VM images on many nodes simultaneously (*multi-deployment*); and (2) efficient concurrent snapshotting of a large number of VM instances to

persistent storage (*multi-snapshotting*). In the context of cloud computing, efficiency means not only fast execution time, but also low network traffic and storage space, as these resources need to be paid by the user proportional to the consumption.

We propose a series of optimization techniques that aim at minimizing both execution time and resource consumption. While conventional approaches transfer the whole VM image contents between the persistent storage service and the computing nodes, we leverage object-versioning to build a lazy deployment scheme that transfers only the needed content on-demand, which greatly reduces execution time and resource consumption. The work presented in this chapter was published in [111].

10.1 Problem definition

The on-demand nature of IaaS is one of the key features that makes it attractive as an alternative to buying and maintaining hardware, because users can rent virtual machines (VMs) instantly, without having to go through lengthy setup procedures. VMs are instantiated from a virtual machine image (simply referred to as image), a file that is stored persistently on the cloud and represents the initial state of the components of the virtual machine, most often the content of the virtual hard drive of the VM.

One of the commonly occurring patterns in the operation of IaaS is the need to instantiate a large number of VMs at the same time, starting from a single (or multiple) images. For example, this pattern occurs when the user wants to deploy a virtual cluster that executes a distributed application, or a set of environments to support a workflow.

Once the application is running, a wide range of management tasks, such as checkpointing and live migration are crucial on clouds. Many such management tasks can be ultimately reduced to snapshotting [154]. This essentially means to capture the state of the running VM inside the image, which is then transferred to persistent storage and later reused to restore the state of the VM, potentially on a different node than the one where it originally ran. Since the application consists of a large number of VMs that run at the same time, another important patten that occurs in the operation of IaaS is concurrent VM snapshotting.

This chapter focuses on highlighting the benefits of BlobSeer for these two patterns. We call these two patterns the *multi-deployment pattern* and the *multi-snapshotting pattern*:

- The *multi-deployment pattern* occurs when multiple VM images (or a single VM image) are deployed on many nodes at the same time. In such a scenario where massive concurrent accesses increase the pressure on the storage service where the images are located, it is interesting to avoid full transfer of the image to the nodes that will host the VMs. At the minimum, when the image is booted, only parts of the image that are actually accessed by the boot process need to be transferred. This saves us the cost of moving the image and makes deployment fast while reducing the risk for a bottleneck on the storage service where images are stored. However, such a "lazy" transfer will make the boot process longer, as some necessary parts of the image may not be available locally. We exploit this tradeoff to achieve a good balance between deployment and application execution.
- The *multi-snapshotting pattern* occurs when many images corresponding to deployed VM instances in a datacenter are persistently saved to a storage system at the same

time. The interesting property of this pattern is that most of the time, only small parts of the image are modified by the VM instance. Therefore, image snapshots share large amounts of data among each other, which can be exploited both to reduce execution time, as well as to reduce storage space and bandwidth consumption.

These two patterns are complementary and for this reason we study them in conjunction.

10.2 Application model

In order to reason about the two patterns presented above, several important aspects need to be modeled.

10.2.1 Cloud infrastructure

Clouds typically are built on top of clusters made out of loosely-coupled commodity hardware that minimizes per unit cost and favors low power over maximum speed [175]. Disk storage (cheap hard-drives with capacities in the order of several hundred GB) is attached to each processor, while processors are interconnected with standard Ethernet links. A part of those nodes is employed as compute nodes that run the VMs of users. Their disk storage is not persistent and is wiped after the VM finished running. Another part of these nodes is employed as storage nodes, which are responsible to host a distributed storage service, such as Amazon S3 [130], whose role is to persistently store both VM images and application data. In many commercial clouds, the ratio of storage nodes to compute nodes is not officially disclosed, but with the recent explosion of data sizes, (for example, Google grew from processing 100 TB of data a day with MapReduce in 2004 [37] to processing 20 PB a day with MapReduce in 2008 [38]), we estimate that the storage nodes will soon have to outnumber the compute nodes to cope with these increasing storage needs.

10.2.2 Application state

The state of the VM deployment is defined at each moment in time by two main components: the state of each of the VM instances and the state of the communication channels between them (opened sockets, in-transit network packets, virtual topology, etc).

Thus, in the most general case (denoted Model 1), saving the application state implies saving both the state of all VM instances and the state of all active communication channels between them. While several methods have been established in the virtualization community to capture the state of a running VM (CPU registers, RAM, state of devices, etc.), the issue of capturing the state of the communication channels is difficult and still an open problem [82]. In order to avoid this issue, the general case is usually simplified such that the application state is reduced to the sum of states of the VM instances (denoted Model 2). However, while this is perfectly feasible for one single VM instance and widely used in practice, for a large number of VMs the necessary storage space explodes to huge sizes. For example, saving 2 GB of RAM for 1000 VMs consumes 2 TB of space, which is unacceptable for a single one-point-in-time application state.

Therefore, Model 2 can further be simplified such that the VM state is represented only by the virtual disk attached to it, which is stored as an image file in the local file system of the VM host (denoted Model 3). Thus, the application state is saved by persistently storing all disk-image files (locally modified by the VM). While this approach requires the application to be able to save and restore its state explicitly to disk (for example as a temporary file), it has two important practical benefits: (1) huge reductions in the size of the state, since the contents of RAM, CPU registers, etc. does not need to be saved; and (2) portability, since the VM can be restored on another host without having to worry about restoring the state of hardware devices that are not supported or are incompatible between different hypervisors.

In this work, for clarity, we assume VM state is represented using Model 3. It is however easy to extend the applicability of our approach to Model 2 by considering that VM image files include not only the contents of the virtual disk attached to the VMs, but also the state of the devices.

10.2.3 Application access pattern

VM typically do not access their whole initial images. For example, they may never access some applications and utilities that are installed by default. In order to model this, it is useful to analyze the life-cycle of a VM instance, which consists of three phases:

1. *Booting*, which involves reading configuration files and launching processes which translates to random small reads and writes from the virtual machine image acting as the initial state.
2. *Running the user application*, which generates application-specific access patterns:
   - *Negligible disk-image access*, which applies to CPU-intensive applications or applications that use external storage services. Examples for this case are large-scale simulations.
   - *Read-your-writes access*, which applies to applications that write temporary files or log files and eventually read them back (e.g., web servers).
   - *Read-intensive access*, which applies to application that read (most often sequentially) input data stored in the image. Examples here are data mining applications. The results are typically presented as a single aggregated value, so generated writes are negligible.
3. *Shutting down*, which generates negligible disk access to the image.

10.3 Our approach

We propose an approach that enables both efficient propagation of the initial virtual image contents to the VMs when the application is deployed, as well as efficient snapshotting while the VM is running.

10.3.1 Core principles

We rely on three key principles:

10.3.1.1 Optimize VM disk access by using on-demand image mirroring

Before the VM needs to be instantiated, an initially empty file of the same size as the image is created on the local file system of the compute node that hosts the VM. This file is then passed to the hypervisor running on the compute node, to be used as the underlying VM image. Read and write accesses to the file however are trapped and treated in a special fashion. A read that is issued on a fully or partially empty region in the file that has not been accessed before (either by a previous read or write), results in fetching the missing content remotely from the repository, creating a local copy of it and redirecting the read to the local copy. If the whole region is available locally, no remote read is performed. This relates closely to *copy-on-reference*, first used for process migration in the V-system [158]. Writes on the other hand are always performed locally.

10.3.1.2 Reduce contention by striping the image

Each virtual image is split into small equally-sized chunks that are distributed among the storage nodes of the repository. When a read request triggers a remote fetch from the repository, the chunks that hold this content are first determined. Then, the storage nodes that hold the chunks are contacted in parallel and the data is transferred back to the compute node. Under concurrency, this scheme effectively enables the distribution of the I/O workload among the storage nodes, because accesses to different parts of the image are served by different storage nodes.

Even in the worst case scenario when all VMs read the same chunks in the same order concurrently (for example, during the boot phase) there is a high chance that the accesses get skewed and thus are not issued at exactly the same time. This effect happens because of various reasons: different hypervisor initialization overhead, interleaving of CPU time with I/O access (which under concurrency leads to a situation where some VMs execute code during the time in which others issue remote reads), etc. For example, when booting 150 VM instances simultaneously, we measured two random instances to have on the average a skew of about 100ms between the times they access the boot sector of the initial image. This skew grows higher the more the VM instances continue with the boot process. What this means is that at some point under concurrency they will access different chunks, which are potentially stored on different storage nodes. Thus, contention is reduced.

Whereas splitting the image into chunks reduces contention, the effectiveness of this approach depends on the chunk size and is subject to a trade-off. A chunk that is too large may lead to false sharing, i.e. many small concurrent reads on different regions in the image might fall inside the same chunk, which leads to a bottleneck. A chunk that is too small on the other hand leads to a higher access overhead, both because of higher network overhead, resulting from having to perform small data transfers and because of higher metadata access overhead, resulting from having to manage more chunks. Therefore, it is important to find the right trade-off.

10.3.1.3 Optimize snapshotting by means of shadowing and cloning

Since our approach does not bring the whole contents of the initial image on the compute node where the VM is running, taking a snapshot is a non-trivial issue. The local image file

cannot be simply saved to persistent storage, as it is partially empty. Even under the assumption that the image contents has been fully pulled on the local file system, it is not feasible to simply save the local image to persistent storage, because a lot of data is duplicated, which leads to unnecessary storage space and bandwidth consumption, not to mention unacceptably high snapshotting times.

For this reason, most hypervisors implement custom image file formats that enable storing incremental differences to efficiently support multiple snapshots of the same VM instance in the same file. For example, KVM introduced the QCOW2 [51] format for this purpose, while other work such as [127] proposes the Mirage Image Format (MIF). This effectively enables snapshots to share unmodified content, which lowers storage space requirements. However, in our context we need to efficiently support multiple snapshots of *different* VM instances that share an initial image. This requirement limits the applicability of using such a custom image file format. Moreover, a custom image file format also limits the migration capabilities: if the destination host where the VM needs to be migrated runs a different hypervisor that does not understand the custom image file format, migration is not possible.

Therefore it is highly desirable to satisfy two requirements simultaneously:

1. store only the incremental differences between snapshots;
2. represent each snapshot as an independent, raw image file that is compatible with all hypervisors.

We propose a solution that addresses these two requirements by leveraging the versioning principles introduced in Section 4.2. In particular, we rely on the update semantics that offers the illusion of creating a new standalone snapshot of the object for each update to it, while physically storing only the differences and manipulating metadata in such way that the aforementioned illusion is upheld. This effectively means that from the user point of view, each snapshot is a *first-class object* that can be independently accessed. For example, let us assume that a small part of a large VM image needs to be updated. With versioning, the user sees the effect of the update as a new, completely independent VM image that is identical to the original except for the updated part. Moreover, we also leverage cloning to duplicate a VM in such way that it looks like a stand-alone copy that can evolve in a different direction than the original, but physically shares all initial content with the original.

Using these two versioning concepts, snapshotting can be easily performed in the following fashion: the first time a snapshot is built, a new virtual image clone is created from the initial image for each VM instance. Subsequent local modifications are written as incremental differences to the clones. This way all snapshots of all VM instances share unmodified content among each other, while still appearing to the outside as independent raw image files.

10.3.2 Applicability in the cloud: model

The simplified architecture of a cloud which integrates our approach is depicted in Figure 10.1. The typical elements found in the cloud are illustrated with a light background, while the elements that are part of our proposal are highlighted by a darker background.

The following actors are present:

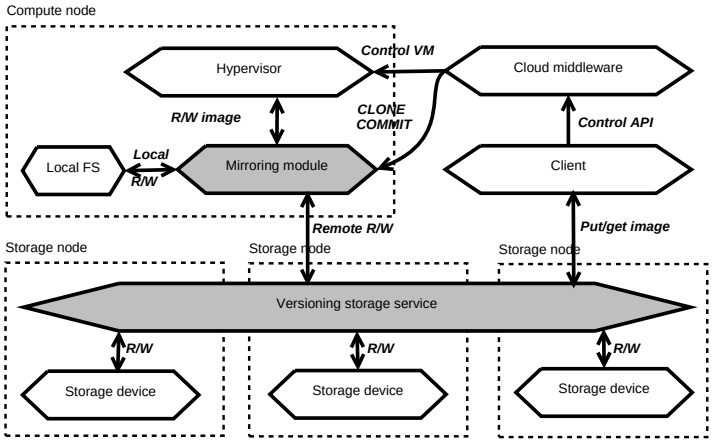


Figure 10.1: Cloud architecture that integrates our approach (dark background)

- *BlobSeer*: in its role as a versioning storage service, it is deployed on the storage nodes and manages their storage devices.
- The *cloud middleware*: it is responsible to expose and implement a control API that enables a wide range of management tasks: VM deployment and termination, snapshotting, monitoring, etc.
- The *cloud client*: it uses the control API of the cloud middleware in order to interact with the cloud. It has direct access to the storage service and is allowed to upload and download images from it. Every uploaded image is automatically striped.
- The *hypervisor*: it runs on the compute nodes and is responsible to execute the VMs. The cloud middleware has direct control over it.
- The *mirroring module*: it traps the reads and writes of the hypervisor and implements on-demand mirroring and snapshotting, as explained in Section 10.3.1. It relies on both the *local file system* and the versioning storage service to do so.

In order to support snapshotting, the mirroring module exposes two control primitives: `CLONE` and `COMMIT`. These two primitives are used according to the procedure described in 10.3.1.3: `CLONE` is used to create a new image clone, while `COMMIT` is used to persistently store the local modifications to it.

Both `CLONE` and `COMMIT` are control primitives that result in the generation of a new fully independent VM image that is globally accessible through the storage service and can be deployed on other compute nodes or manipulated by the client. A global snapshot of the whole application, which involves taking a snapshot of all VM instances in parallel, is performed in the following fashion: the first time when the snapshot is taken, `CLONE` is broadcast to all mirroring modules, followed by `COMMIT`. Once a clone is created for each VM instance, subsequent global snapshots are performed by issuing to each mirroring module a `COMMIT` to its corresponding clone.

`CLONE` and `COMMIT` can also be exposed by the cloud middleware at user level through the control API for fine-grain control over snapshotting. This enables leveraging snapshotting

in interesting ways. For example, let us assume a scenario where a complex distributed application needs to be debugged. Running the application repeatedly and waiting for it to reach the point where the bug happens might be prohibitively expensive. However, `CLONE` and `COMMIT` can be used to capture the state of the application right before the bug happens. Since all virtual image snapshots are independent entities, they can be either collectively or independently analyzed and modified in an attempt to fix the bug. Once this is done, the application can safely resume from the point where it left. If the attempt was not successful, this can continue iteratively until a fix is found. Such an approach is highly useful in practice at large scale, because complex synchronization bugs tend to appear only in large deployments and are usually not triggered during the test phase, which is usually performed at smaller scale.

10.3.3 Zoom on mirroring

One important aspect of on-demand mirroring is the decision of how much to read from the repository when data is unavailable locally, in such way as to obtain a good access performance.

A straightforward approach is to translate every read issued by the hypervisor in either a local or remote read, depending whether the requested contents is locally available or not. While this approach certainly works, its performance is rather questionable. More specifically, many small remote read requests generate significant network traffic overhead (because of the extra networking information encapsulated with each request), as well as a low throughput (because of the latencies of the requests that add up). Moreover, in the case of many scattered small writes, a lot of small fragments need to be accounted for, in order to remember what is available locally for reading and what is not. A lot of fragments however incur a significant management overhead, negatively impacting access performance.

For this reason, we leverage two heuristics that aim to limit the negative impact of small reads and writes. The first heuristic is a prefetching heuristic that is based on the empirical observation that reads tend to be locally correlated: a read on one region is probably followed by a read "in the neighborhood". This is especially true for sequential read access, which is a common access pattern. For this reason, the heuristic tries to minimize the negative impact of small reads by forcing remote reads to be at least as large as a chunk. Subsequent reads that fall within the same chunk are served locally, thus greatly improving throughput in this case.

The second heuristic we propose eliminates write fragmentation by forcing a single contiguous region to be mirrored locally for each chunk. More precisely, when a chunk that is not available locally is written for the first time, only the region that is covered by the write is modified in the local file. If a subsequent write to the same chunk falls on a region that is disjoint from the first write, a gap between the two regions is created. In this case, the heuristic reads the region corresponding to the gap from the remote storage and applies it to the local file, such that a single contiguous non-empty region is obtained in the local file. With this approach, only the limits of the contiguous region need to be maintained for each chunk, which makes fragment management overhead negligible. Moreover, it is better than plain copy-on-reference which would read the whole chunk before applying a write, because it avoids unnecessary reads when writes are not fragmented (e.g., sequential writes).

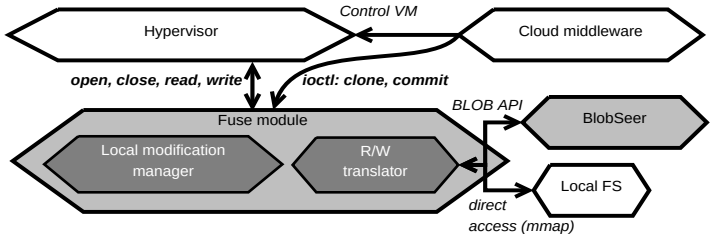


Figure 10.2: Implementation details: zoom on the FUSE module

10.4 Implementation

In this section we show how to efficiently implement the building blocks presented in the previous section in such way that they achieve the design principles introduced in Section 10.3.1 on one side, and are easy to integrate in the cloud on the other side.

The versioning storage service relies on *BlobSeer*, which brings three advantages in this context. First it offers out-of-the-box support for versioning, which enables easy implementation of our approach. Second, BlobSeer supports transparent data striping of large objects and fine-grain access to them, which enables direct mapping between BLOBs and virtual machine images and therefore eliminates the need for explicit chunk management. Finally it offers support for high throughput under concurrency, which enables efficient parallel access to the image chunks.

The mirroring module was implemented on top of *FUSE* (FileSystem in UserspacE) [176]. FUSE is a loadable kernel module for Unix-like operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module acts only as an intermediate to the actual kernel interfaces. This approach brings several advantages in our context. First, it enables portability among hypervisors by exposing a POSIX-compliant file system access interface to the images. POSIX is supported by most hypervisors and enables running the same unmodified mirroring module on all compute nodes, regardless of what hypervisor is installed on them. Second, FUSE takes advantage of the kernel-level virtual file system, which brings out-of-the-box support for advanced features such as cache management. Finally it avoids the need to alter or extend the hypervisor in any way, which effectively reduces implementation time and maintenance costs. The downside of FUSE is the extra context switching overhead between the kernel space and the user space when an I/O system call is issued. However, this overhead has a minimal negative impact, as demonstrated in Section 10.5.3.

BlobSeer is leveraged as a versioning storage service by mapping each virtual image to a BLOB. Local modifications are committed simply by using the BlobSeer write primitive to update the BLOB. Cloning is performed by using the BlobSeer clone primitive.

The FUSE module, presented in Figure 10.2 exposes each BLOB (the VM) as a directory in the local file system, and its associated snapshots as files in that directory. It consists of two sub-modules: the *local modification manager*, responsible to track what contents is available locally and the *R/W translator*, responsible to translate each original read and write request into local and remote reads and writes, according to the strategy presented in Section 10.3.3.

Whenever a BLOB snapshot (exposed by FUSE as a file to the hypervisor) is opened for the first time, an initially empty file of the same size as the BLOB snapshot is created on the local filesystem. This file is then used to mirror the contents of the BLOB snapshot. As soon as this operation completed, the local file is *mmap*-ed in the host's main memory for as long as the snapshot is still kept open by the hypervisor.

Reads and writes issued by the hypervisor are trapped by the R/W translator and broken by the local modification manager into elementary read and write operations that concern either the remote storage service or the local file system. Thanks to mmap, any local read and write operation bypasses the POSIX interface and is executed as a direct memory access. This enables *zero-copy* (i.e. avoiding unnecessary copies between memory buffers). Moreover, local writes are also optimized this way because they benefit from the built-in asynchronous mmap write strategy.

Finally, when the snapshot is closed, the mmapped space is unmapped and the local file is closed.

`CLONE` and `COMMIT` are implemented as *ioctl* system calls. Both rely on the clone and write primitives natively exposed by BlobSeer. `CLONE` simply calls the BlobSeer clone primitive and binds all local modifications to the newly cloned BLOB, while the `COMMIT` primitive writes all local modifications back to the BLOB as a series of BlobSeer writes. Care is taken to minimize the amount of issued BlobSeer writes by aggregating consecutive "dirty" (i.e. locally modified) chunks in the same write call. Once the all dirty chunks have been successfully written, the state is reset and all chunks are marked as "clean" again.

For the purpose of this work, we did not integrate the `CLONE` and `COMMIT` primitives with a real cloud middleware. We implemented a simplified service instead that is responsible to coordinate and issue these two primitives in a series of particular experimental scenarios that are described in the next section.

10.5 Evaluation

10.5.1 Experimental setup

The experiments presented in this work have been performed on Grid'5000 [68], using the cluster located in Nancy. All nodes of Nancy, numbering 200 in total, are outfitted with x86_64 CPUs, local disk storage of 250 GB (access speed 55 MB/s) and at least 2 GB of RAM. Unless otherwise stated, we fix 50 nodes to act as storage nodes and we employ a variable number of compute nodes, up to the rest of 150. This accounts for a storage-to-compute ratio that is at least 1:3. The hypervisor running on all compute nodes is KVM 0.12. For all experiments, a 2 GB raw disk image file based on a recent Debian Sid distribution was used.

10.5.2 Scalability of multi-deployment under concurrency

The first series of experiments evaluates how well our approach performs under the multi-deployment pattern, when a single initial image is used to instantiate a large number of VM instances.

We compare our approach to the technique commonly used by cloud middleware to distribute the image content on the node that hosts the VM: *pre-propagation*. Pre-propagation

consists of two phases: in a first phase the VM image is broadcast to the local storage of all compute nodes that will run a VM instance. Once the VM image is available locally on all compute nodes, in the second phase all VMs are launched simultaneously. Since in the second phase all content is available locally, no remote read access to the repository is necessary. This enables direct local access to the image and does not depend or use the network connection to the storage nodes. This is based on the assumption that access to the local file system of the compute nodes is faster than access to remote storage, which is the case for most large clusters in the industry that are built from commodity hardware, as mentioned in Section 2.1.1.

The downside of this approach is however the initialization phase, which potentially incurs a high overhead, both in terms of latency and network traffic. In order to minimize this overhead, we use TakTuk [33], a highly scalable tool based on broadcasting algorithms in the postal model [12], which builds adaptive multicast trees that optimize the bandwidth/latency tradeoff in order to efficiently broadcast a file to a set of machines. In this setting, it is assumed that the root of the multicast tree is a NFS file server which is hosted by a dedicated storage node and which holds the initial image.

We will compare the two approaches according to the following metrics:

- *Average execution time per instance*: the average time taken to execute an application in the VM. This time is measured after the initialization phase (if applicable) has been completed, between the time the VM is launched and the time the VM is terminated. This parameter is relevant because it reveals the impact of remote concurrent reads (present in our approach) vs. independent local reads (pre-propagation) on the scalability of running VMs in parallel.

- *Time-to-completion for all instances*: the time taken to complete the initialization, launching, and execution of applications for all VMs. This time is measured between when the request to launch the VMs is received in the system and when the last VM is terminated. This parameter is relevant because it measures the total time needed to execute the application and obtain a final result, which is what the user directly perceives.

- *Total network traffic*: the total network traffic generated throughout the execution of all VMs, including during the initialization phase (if applicable). This parameter is relevant because it directly impacts the end-user costs.

The series of experiments consists in deploying an increasing number of VMs using both our approach and pre-propagation. We start with one VM and increase the number of deployed VMs in steps of 25 up to 150, applying the metrics defined above at each step. In the case of pre-propagation, the initial image is stored on the local NFS server which serves the cluster and which is used as the source of the multicast. In the case of our approach, BlobSeer is assumed to be deployed on the 50 reserved storage nodes and the initial image stored in a striped fashion on it.

Note that the variation of average execution time needs not be represented explicitly. This results from two reasons: (1) in the case of pre-propagation all data is available locally after the initialization phase and therefore the variation is negligible; and (2) in the case of our approach there is no initialization phase and therefore the completion time coincides with the time to run the slowest VM, which measures the maximal deviation from the mean.

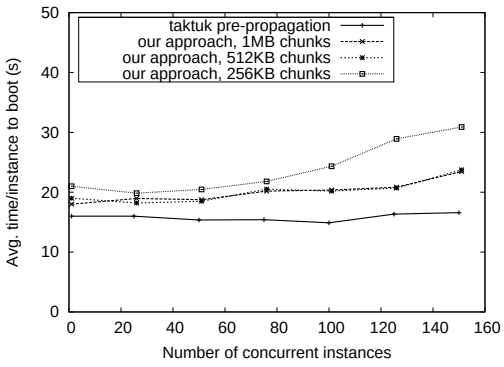


Figure 10.3: Average time to boot per instance

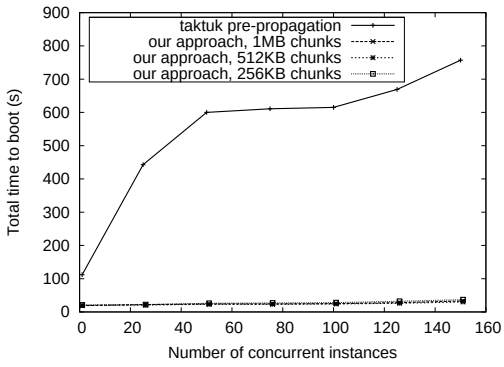


Figure 10.4: Completion time to boot all instances

10.5.2.1 Boot performance

In the first experiment, the only activity carried out in the VM is fully booting the operating system. This corresponds to the behavior of an application that performs minimal access to the underlying virtual image, which is for example the case of CPU-intensive applications. Under such circumstances, almost all access to the virtual image is performed during the boot phase.

Since in our approach the image is striped into chunks, an important aspect is to evaluate the impact of the chunk size on performance and network traffic. As discussed in Section 10.3.1.2, a small chunk size minimizes the amount of unnecessary data that is prefetched and thus minimizes network traffic. However, lowering the chunk size too much means that many chunks have to be fetched and thus this can have a negative impact on performance. We evaluated various chunk sizes and found the best performance was delivered at 512 KB. For completeness, we include both results obtained with a higher size (1 MB), and a lower size (256 KB).

Figure 10.3 shows the average boot time per VM instance. As expected, in the case of pre-propagation, average boot time is almost constant, as data is already on the local file system and therefore no transfer from the storage nodes is required. In the case of our approach, boot times are higher, as chunks need to be fetched remotely from the storage nodes on-the-fly during boot time. The more instances, the higher the read contention and thus the higher the boot times. A breaking point is noticeable at 50 instances, where the increase in average time becomes steeper. This is because after 50 instances, the storage nodes are outnumbered by the VM instances and therefore the probability of concurrent access to the same storage node increases even when reading different chunks.

Figure 10.4 shows the total time to boot all VMs. As can be seen, the pre-propagation is an expensive step, especially when considering that only a small part of the initial virtual is actually accessed. This brings our approach at a clear advantage, with the speedup depicted in Figure 10.5. The speedup is obtained as the completion time to boot all instances of the pre-propagation approach divided by the completion time of our approach. As expected, the highest speedup is obtained when the number of VM instances is the same as then number of storage nodes, which enables the optimal distribution of I/O workload for our approach. Performance-wise, it can be observed that a 512 KB chunk size brings the highest speedup

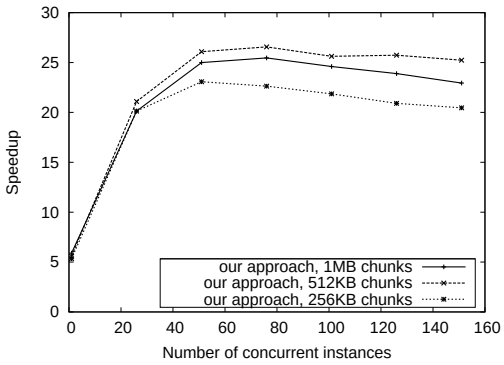


Figure 10.5: Performance of VM boot process: speedup of vs. pre-propagation

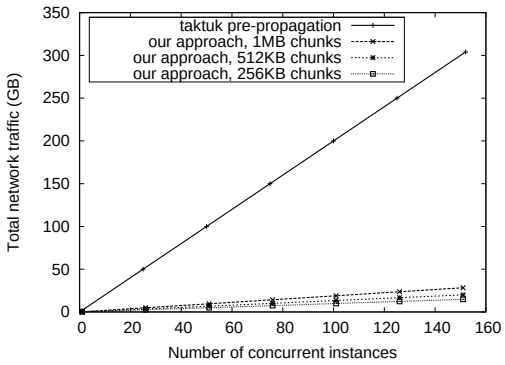


Figure 10.6: Total network traffic generated by our approach vs. multi-cast

and scales well even after the breaking point is reached.

Finally, Figure 10.6 illustrates the total network traffic generated by both approaches. As expected, the growth is linear and is directly proportional to the amount of data that was brought locally on the compute node. In the case of pre-propagation, the network traffic is as expected a little over 300 GB for 150 instances. In the case of our approach, the smaller the chunk size, the smaller amount of total network traffic. However, the total amount of network traffic is not directly proportional to the chunk size. Lowering the chunk size from 1 MB to 512 KB results in a network traffic drop from 30 GB to 21 GB, while lowering from 512 KB to 256 KB in a drop from 21 GB to 15 GB only. The smaller the chunk gets, the smaller the benefits from avoiding network traffic are. This happens mainly because of access locality: consecutive reads issued by the hypervisor fall inside a region that is more likely to be covered by consecutive chunks, which makes the benefit of small chunks sizes smaller.

10.5.2.2 Full read performance

The second experiment considers the complementary case to the one presented in the previous section, namely when the whole content of the virtual image needs to be read by each VM instance. This represents the most unfavorable read-intensive scenario that corresponds to applications which need to read input data stored in the image. In this case, the time to run the VM corresponds to the time to boot and fully read the whole virtual image disk content (by performing a "cat /dev/hda1", where hda1 is the virtual disk corresponding to the image). Again, the evaluation is performed for three chunk sizes: 1 MB, 512 KB and 256 KB.

The average time to boot and fully read the initial disk content is represented in Figure 10.7. As expected, in the case of pre-propagation, this time remains constant as no read contention exists. In the case of our approach, almost perfect scalability is also noticeable up to 50 storage nodes, despite read concurrency. This is so because there are enough storage providers among which the I/O workload can be distributed. After the number of instances outnumbers the storage nodes, the I/O pressure increases on each storage node, which makes the average read performance degrade in a linear fashion. On a general note, the read performance is obviously worse in or approach, as the data is not available locally.

However, when considering the completion time for booting and fully reading the image

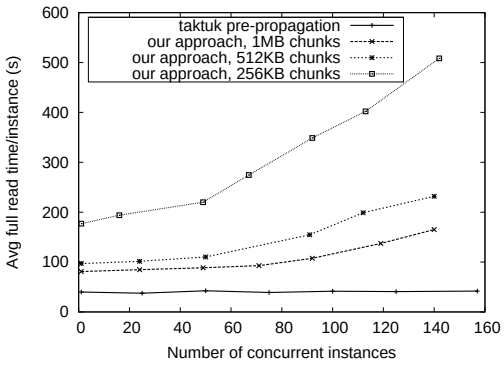


Figure 10.7: Average time to boot and fully read image per instance

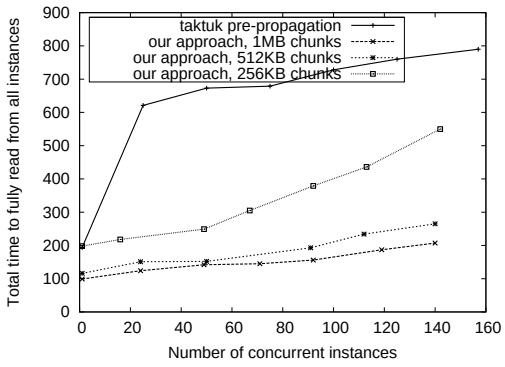


Figure 10.8: Completion time to boot and fully read the contents of all images

from all instances 10.8, our approach is at a clear advantage. The main reason for this is the fact that the pre-propagation approach needs to touch the data twice: once in the initialization phase, when it needs to be copied locally and then in the execution phase when it needs to be read by the guest. Since in our approach the initialization phase is missing, the total time is better even for a single instance.

The actual speedup is represented in Figure 10.9. The peak is as expected at 50 when the number of instances reaches the number of storage nodes. Then, the speedup gradually starts falling as the I/O pressure on the storage nodes increases. Since the image needs to be fully read by each instance, the total generated network traffic is not represented explicitly, as it is roughly the same for both approaches and was already depicted in Figure 10.6: it corresponds to the pre-propagation curve. Compared to the speedup obtained for the boot-only experiment, it can be noted that the peak speed-up is much lower at 4.7 and degrades faster when the number of compute nodes outnumber the storage nodes. This is explained by the fact that in the boot-only experiment, our approach had the advantage of fetching only the necessary parts of the image, which does not apply any longer.

10.5.3 Local access performance: read-your-writes access patterns

While the previous section evaluates first time read performance of the initial image contents, this section focuses on local read and write performance, i.e. when the accessed regions are presumed to be already available locally. This scenario is representative of read-your-writes applications, that need to write large amounts of data in the virtual image (e.g., log files) and then eventually read all this information back.

For this purpose, we compare our approach to the ideal scenario: when a copy of the virtual image is already available on the local file system of the compute node and can be used directly by the hypervisor. We aim to evaluate the overhead of our approach which needs to trap reads and writes and needs to manage local modifications, as opposed to the ideal case when the hypervisor can directly interact with the local file system. In order to generate a write-intensive scenario that also reads back written data, we use a standard benchmarking tool: Bonnie++ [90]. Bonnie++ creates and writes a set of files that fill a large part of the remaining free space of the disk, then reads back the written data, and then

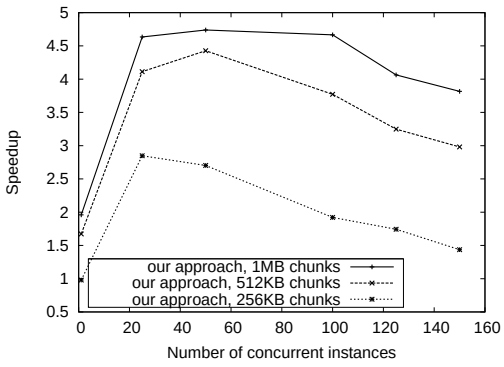


Figure 10.9: Performance of boot followed by full read: speedup vs. pre-propagation

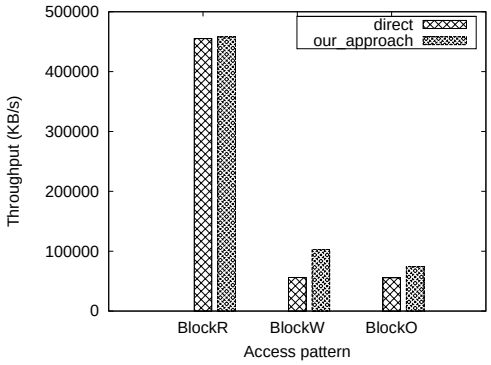


Figure 10.10: Bonnie++ sustained throughput: read, write and overwrite in blocks of 8K (BlockR/BlockW/BlockO)

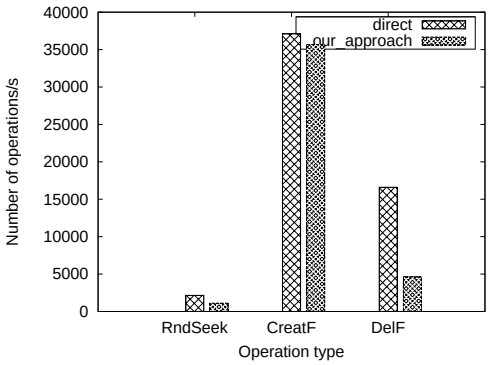


Figure 10.11: Bonnie++ sustained number of operations per second: random seeks (RndSeek), file creation/deletion (CreatF/DelF)

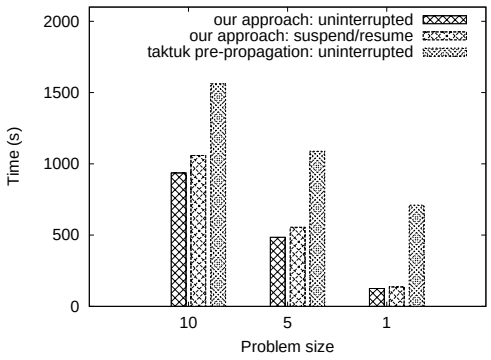


Figure 10.12: Monte-Carlo PI calculation: time to finish estimating PI using 100 instances that each pick 1, 5 and 10 billion points

overwrites the files with new data, recording throughput in all cases. Other performance factors such as how many files per second can be created and deleted are also recorded. Since data is first sequentially written and then read back, no remote reads are involved for our approach. This in turn means contention is not an issue and therefore experimentation with a single VM instance is enough to predict behavior of multiple instances that run concurrently.

The experiment consists in booting the VM instance and then running Bonnie++ using both our approach and a locally available image directly. This experiment is repeated 5 times and the results of Bonnie++ are averaged. The total space written and read back by Bonnie++ was 800 MB out of a total of 2 GB, in blocks of 8 KB.

Throughput results are shown in Figure 10.10. As can be noticed, reads of previously written data have the same performance levels for both approaches. This results is as expected, because previously written data is available locally for both approaches and therefore no additional overhead is incurred by our approach. Interestingly enough, write throughput and overwrite throughput is almost twice as high for our approach. This is

explained by the fact that mmap triggers a more efficient write-back strategy in the host's kernel and overrides the default hypervisor strategy, which comes into play when it has direct access to the image.

On the other hand, the extra context switches and management overhead incurred by our approach become visible when measuring the number of operations per second. Figure 10.11 shows lower numbers for our approach, especially with random seeks and file deletion. However, since operations such as file creation/deletion and seeks are relatively rare and execute very fast, the performance penalty in real life is negligible.

10.5.4 Multi-snapshotting performance

This section evaluates the performance of our approach in the context of the multi-snapshotting access pattern. We assume a large number of VM instances that need to concurrently save their corresponding VM image, which suffered local modifications, persistently on the storage nodes.

The experimental setup is similar to the one used in the previous sections: an initial RAW image, 2 GB large, is striped and stored on 50 storage nodes in three configurations: 1 MB, 512 KB and 256 KB chunks. In order to underline the benefits our our approach better, we consider that local modifications are small, which is consistent for example with checkpointing a CPU-intensive, distributed applications where the state of each VM instance can be written as a temporary file in the image. In order to simulate this behavior in our experiments, we have instantiated a variable number of VM instances from the same initial image and then instructed each instance to write a 1 MB file in its corresponding image, after which the hypervisor was instructed to flush all local modifications to the VM image. Once all hypervisors finished this process, all images were concurrently snapshotted by broadcasting a `CLONE`, followed by a `COMMIT` command to all compute nodes hosting the VMs. The local modifications captured by `COMMIT` include not only the temporary file, but also file system operations performed during the boot phase (i.e. creating configuration files, writing to log files, etc.)

The execution time on each compute node, as well as the total storage space consumed by all snapshots is recorded and used to calculate the following metrics: the average snapshotting time per instance, the completion time to snapshot all VMs, and the overall storage space occupied on the storage nodes (which is roughly equivalent to the total generated network traffic).

Execution time results are depicted in Figure 10.13 and Figure 10.14. As can be observed, average snapshotting time does not show a significant growth when the storage nodes outnumber the compute nodes, and from that point on it shows a linearly-shaped growth. The completion time to snapshot all images follows the same trend, but it grows faster and experiences an evolution in steps. This is explained by the higher deviation from the mean that is present from the increased write pressure on the storage nodes. Nevertheless, both the average time as well as the completion time are just a tiny fraction when compared to traditional approaches that store all images fully in the cloud storage, a process that can easily extend to hours. The same conclusion can be drawn when looking at the storage space and network traffic: Figure 10.15 shows a total occupied storage space well below 2 GB for 150 instances, when the chunk size is 256 KB, and about 4.5 GB for a 1 MB chunk size. For comparison, fully transferring the images to the storage nodes takes well over 300 GB for 150 instances.

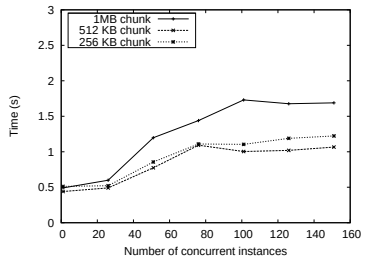


Figure 10.13: Average time to snapshot an instance under concurrency

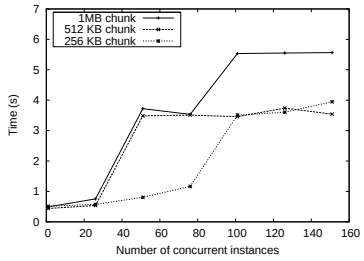


Figure 10.14: Completion time to snapshot all instances concurrently

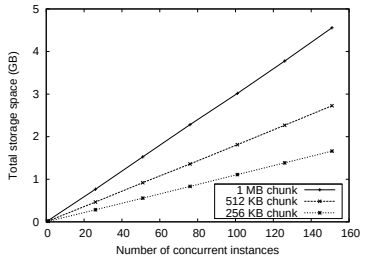


Figure 10.15: Total aggregated storage space occupied by all snapshots

10.5.5 Benefits for real-life, distributed applications

We illustrate the benefits of our approach for a common type of CPU-bound scientific applications: Monte-Carlo approximations. Such applications rely on repeated random sampling to compute their results and are often used in simulating physical and mathematical systems. This approach is particularly useful when it is unfeasible or impossible to compute the exact result using deterministic methods. In our particular case, we estimate the number π by choosing random points in a square and calculating how much of the points fall inside of the inscribed circle. In this case, $\pi = 4 \times (points\_inside)/(total\_points)$.

This is an embarrassingly parallel application, as a large number of workers can independently pick such points and verify their belonging to the circle. In a final step, the results are aggregated and π is calculated. We spawn 100 instances to solve this problem for a variable number of total points: 10^{11}, 5×10^{11} and 10^{12}. The work is evenly distributed among instances. The workload is mostly CPU-intensive, with each instance programmed to save intermediate results at regular intervals in a temporary file (I/O is negligible).

For each total number of points, the problem is solved in three different ways: using our approach, using pre-propagation as described in Section 10.5.2, and using our approach but suspending the application and then resuming each VM instance on a different node as where it originally ran. The suspend-resume is performed in the following fashion: `CLONE`, followed by a `COMMIT`, is broadcast to all VM instances. Immediately after all snapshots are successfully written to persistent storage, the VM instances are killed. Once all VM instances are down, they are rebooted and the application resumes from the last intermediate result saved in the temporary file. Each VM instance is resumed on a different compute node than the one where it originally ran, to simulate a realistic situation where the original compute nodes have lost all their local storage contents or there is a need to migrate the whole application on a different set of compute nodes.

Results are shown in Figure 10.12. As expected from the results obtained in Section 10.5.2.1, since this is a CPU-intensive application, the initialization phase for pre-propagation is extremely costly. This effect is more noticeable when fewer work is done by each VM instance. While the overhead of snapshotting is negligible in all three cases, the suspend-resume cycle overall grows slightly higher the more work needs to be performed by the VM instances, as resuming on different nodes with potentially different CPU creates a larger variance in longer VM execution times. However, this accounts for less than 10% overall overhead in the worst case of 10^{12} points.

10.6 Positioning of this contribution with respect to related work

Several mechanisms to disseminate the content of the initial virtual image exist. Commonly in use is full virtual image pre-propagation to the local storage of the compute nodes before launching the VM. For this purpose, efficient broadcast techniques have been proposed. TakTuk [33] is a highly scalable tool inspired by broadcasting algorithms in the postal model [12]. It builds adaptive multi-cast trees to optimally exploit bandwidth and latency for content dissemination. Multi-casting is also employed by Frisbee [61], a tool used by EmuLab [60] to apply disk images to nodes. Scp-wave [177], in use by OpenNebula [101] is another such tool. It carries out the transfers by starting with a single seed and increases the number of seeders as more content is transferred, in order to obtain a logarithmic speed-up versus a sequential transfer. A similar idea is implemented in [132] as well. While these approaches avoid read contention to the repository, they can incur a high overhead both in network traffic and execution time, as presented in Section 10.5.2.

A different approach to instantiate a large number of VMs from the same initial state is proposed in [74]. The authors introduce a new cloud abstraction: VM `FORK`. Essentially this is the equivalent of the fork call on UNIX operating systems, instantaneously cloning a VM into multiple replicas running on different hosts. While this is similar to `CLONE` followed by `COMMIT` in our approach, the focus is on minimizing the time and network traffic to spawn and run on-the-fly new remote VM instances that share the same local state of an already running VM. Local modifications are assumed to be ephemeral and no support to store the state persistently is provided.

Closer to our approach is Lithium [57], a fork-consistent replication system for virtual disks. Lithium supports instant volume creation with lazy space allocation, instant creation of writable snapshots, and tunable replication. While this achieves the same as `CLONE` and `COMMIT`, it is based on log-structuring [136], which incurs high sequential read and maintenance overhead.

Content Addressable Storage (CAS) [63, 123] was also considered for archival of virtual machine images [103] and disk image management [80, 128]. While the focus is on providing space-efficient disk image storage mainly by exploiting duplication, concurrency issues are again not addressed or not part of the original design.

Cluster volume managers for virtual disks such as Parallax [96] enable compute nodes to share access to a single, globally visible block device, which is collaboratively managed to present individual virtual disk images to the VMs. While this enables efficient frequent snapshotting, unlike our approach, sharing of images is intentionally unsupported in order to eliminate the need for a distributed lock manager, which is claimed to dramatically simplify the design.

Several storage systems such as Amazon S3 [130] (backed by Dynamo [39]) have been specifically designed as highly available key-value repositories for cloud infrastructures. This objective however is achieved at the cost of limiting the client to read and write full objects only, which limits the applicability in our context.

Finally, our approach is intended as a means to complement existing cloud computing platforms, both from the industry (Amazon Elastic Compute Cloud: EC2 [175]) and academia (Nimbus [71], OpenNebula [101]). While the details for EC2 are not publicly available, it is widely acknowledged that all these platforms rely on several of the techniques

presented above. Claims to instantiate multiple VMs in "minutes" however is insufficient for the performance objectives of our work.

10.7 Conclusions

In this chapter we have successfully demonstrated the benefits of BlobSeer for VM management in the context of IaaS cloud computing. We introduced several techniques that integrate with cloud middleware to efficiently handle two patterns: *multi-deployment* and *multi-snapshotting*.

We proposed a lazy VM deployment scheme that fetches VM image content as needed by the application executing in the VM, thus reducing the pressure on the VM storage service for heavily concurrent deployment requests. Furthermore, we leverage object-versioning to save only local VM image differences back to persistent storage when a snapshot is created, yet provide the illusion that the snapshot is a different, fully independent image. This has an important benefit in that it handles the management of updates independently of the hypervisor, thus greatly improving the portability of VM images, and compensating for the lack of VM image format standardization.

We demonstrated the benefits of the proposal through experiments on 100s of nodes using benchmarks as well as real-life applications. Compared to simple approaches based on pre-propagation, our approach shows improvements both in execution time and resources usage (i.e., storage space and network traffic): we show that the time to process application execution in an IaaS cloud can be improved by a factor of up to 25, while at the same time reducing storage and bandwidth usage by 90%. These results have impact on the final user costs, as costs are directly proportional to the amount of consumed resources and the time the user spends waiting for an application to finish.

Chapter 11

Quality-of-service enabled cloud storage

Contents

In the previous chapter we have evaluated BlobSeer in its role as a storage service that offers efficient support for virtual machine storage in the context of IaaS cloud computing. We have experimented with applications that are rather compute-intensive and have modest user data storage requirements: it is feasible to store all user data in the guest filesystems of the VM instances (i.e. directly in the image files).

However, for data-intensive applications, an external storage service that is accessible from within the VM instances is a more robust approach. This greatly simplifies data management for the user, because input and output files of applications can be uploaded and downloaded by direct interaction with the storage service, without the need to involve virtual machine images.

In this chapter, we evaluate BlobSeer as a cloud storage service for user data rather than virtual machine images. To place this work in a relevant context, we have chosen to use BlobSeer as a storage service for MapReduce applications. However, unlike Chapter 9, where we focused on delivering a *high aggregated throughput* even under a heavy data access concurrency, in this chapter we focus on a crucial property on clouds: *quality-of-service* (QoS).

QoS is important in this context, because the cloud storage service is shared by multiple users and therefore a high aggregated throughput does not necessarily imply that all customers are served equally well. Therefore, in addition to sustaining a high aggregated throughput, in this chapter we aim to deliver a *stable throughput for each individual data access*. A stable throughput guarantee is however an inherently difficult task, because a large number of factors is involved.

First, MapReduce frameworks run on infrastructures comprising thousands of commodity hardware components. In this context, failures are rather the norm than the exception. Since faults cause drops in performance, *fault tolerance* becomes a critical aspect of throughput stability. Second, complex data access patterns are generated that are likely to combine periods of intensive I/O activity with periods of relative less intensive I/O activity throughout run-time. This has a negative impact on throughput stability, thus *adaptation to access pattern* is a crucial issue as well.

The sheer complexity of both the state of the hardware components and the data access pattern makes reasoning about fault tolerance and adaptation to access pattern difficult, since it is not feasible to find non-trivial dependencies manually. Therefore, it is important to automate the process of identifying and characterizing the circumstances that bring the storage service in a state where there are significant fluctuations in the sustained throughput and to take appropriate action to stabilize the system.

In order to solve this problem, we propose a general methodology based on *component monitoring*, *application-side feedback* and *behavior pattern analysis* in order to discover and characterize the situations that lead to fluctuations of individual data access throughput and remedy the situation. This work was published in [98].

11.1 Proposal

We propose a general approach to automate the process of identifying and characterizing the events that cause significant fluctuations in the sustained throughput of individual I/O transfers. This enables the storage service to take appropriate action in order to stabilize the system. We rely on three key principles.

In-depth monitoring. The storage service is usually deployed on large-scale infrastructures comprising thousands of machines, each of which is prone to failures. Such components are characterized by a large number of parameters, whose values need to be measured in order to describe the state of the system at a specific moment in time. Since data-intensive applications generate complex access-pattern scenarios, it is not feasible to predetermine which parameters are important and should be monitored, as this limits the potential to identify non-obvious bottlenecks. Therefore, it is important to collect as much information as possible about each of the components of the storage service.

Application-side feedback. Extensive monitoring of the system helps to accurately define its state at a specific moment in time. However, it is not accurate enough to identify a situation where the throughput of individual I/O transfers fluctuates, because the perceived quality of service from the application point of view remains unknown. For example, in the context of MapReduce applications, monitoring the network traffic is not enough to infer the throughput achieved for each task, because the division of each job into tasks remains unknown to the storage service. Therefore, it is crucial to gather feedback dynamically from the upper layers that rely on the storage service in order to decide when the system performs satisfactorily and when it does not.

Behavior pattern analysis. Extensive monitoring information gives a complete view of the behavior of the storage service in time. Using the feedback from the upper layers, it is also possible to determine when it performed satisfactory and when it performed poorly. However, the complexity of the behavior makes direct reasoning about the causes of potential bottlenecks difficult. Intuitively, explaining the behavior is much easier if a set of behavior patterns can be identified, which can be classified either as satisfactory or not. Once a classification is made, taking action to stabilize the system essentially means to predict a poor behavior and enforce a policy to avoid it. The challenge however is to describe the behavior patterns in such a way that they provide meaningful insight with respect to throughput stability, thus making healing mechanisms easy to implement.

11.1.1 Methodology

Starting from these principles, we propose a methodology to stabilize the throughput of the storage service as a series of four steps:

Monitor the storage service. A wide range of parameters that describe the state of each of the components of the storage service is periodically collected during a long period of service up-time. This is necessary in order to approximate reliably the data access pattern generated by the data-intensive application. An important aspect in this context is *fault detection*, as information on when and how long individual faults last is crucial in the identification of behavior patterns.

Identify behavior patterns. Starting from the monitored parameters, the behavior of the storage service as a whole is classified into behavior patterns. The classification must be performed in such a way that, given the behavior at any moment in time, it unambiguously matches one of the identified patterns. The critical part of this step is to extract meaningful information with respect to throughput stability that describes the behavior pattern. Considering the vast amount of monitoring information, it is not feasible to perform this step manually. Therefore, an automated knowledge discovery method needs to be applied. Specific details are provided in Section 11.1.2.

Classify behavior patterns according to feedback. In order to identify a relationship between behavior patterns and stability of throughput, the application-side feedback with respect to the perceived quality of service is analyzed. More specifically, a series of performance metrics as observed by the upper layers is gathered for each data

transfer. These performance metrics are then aggregated for all data transfers that occurred during the period in which the behavior pattern was exhibited. The result of the aggregation is a score that is associated to the behavior pattern and indicates how desirable the behavior pattern is. Once the classification has been performed, transitions from desirable states to undesirable states are analyzed in order to find the reason why the storage service does not offer a stable throughput any longer. This step involves user-level interpretation and depends on the quality of the generated behavior model.

Predict and prevent undesired behavior patterns. Finally, understanding the reasons for each undesired behavior enables the implementation of prediction and prevention mechanisms accordingly. More specifically, the preconditions that trigger a transition to each undesirable state are determined. Using this information, a corresponding policy is enabled, capable of executing a special set of rules while these preconditions are satisfied. These rules are designed to prevent this transition to occur or, if this is not possible, to mitigate the effects of the undesired state. It is assumed that the application is monitored throughout its execution and the storage service has accessed to the monitoring information. This makes the service able to predict when an undesirable behavior is about to happen and to activate the corresponding policy.

11.1.2 GloBeM: Global Behavior Modeling

In order to identify and describe the behavior patterns of the storage service in a simple yet comprehensive way, we rely on a generic approach to model the global behavior of large-scale distributed systems [99, 100] (from now referred to as *GloBeM*). Its main objective is to build an abstract, descriptive model of the global system state. This enables the model to implicitly describe the interactions between entities, which has the potential to unveil non-trivial dependencies significant for the description of the behavior, which otherwise would have gone unnoticed.

GloBeM follows a set of procedures in order to build such a model, starting from monitoring information that corresponds to the observed behavior. These basic monitoring data are then aggregated into *global monitoring parameters*, representative of the global system behavior instead of each single resource separately. This aggregation can be performed in different ways, but it normally consists in calculating global statistic descriptors (mean, standard deviation, skewness, kurtosis, etc.) values of each basic monitoring parameter for all resources present. This ensures that global monitoring metrics are still understandable from a human perspective. This global information undergoes a complex analysis process in order to produce a global behavior representation. This process is strongly based on machine learning and other knowledge discovery techniques, such as virtual representation of information systems [161, 162]. A behavior model presents the following characteristics.

Finite state machine. The model can be expressed as a finite state machine, with specific states and transitions. The number of states is generally small (between 3 and 8).

State characterization based on monitoring parameters. The different system states are expressed in terms of the original monitoring parameters. This ensures that its characteristics can be understood and directly used for management purposes.

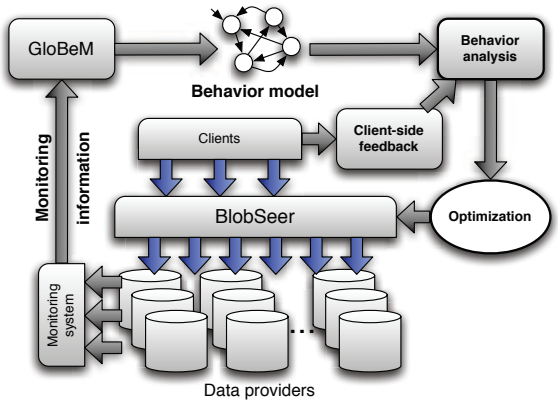


Figure 11.1: Our approach applied: stabilizing throughput in BlobSeer

Extended statistical information. The model is completed with additional statistic metrics, further expanding the state characterization.

11.1.3 Applying the methodology to BlobSeer

The methodology described in Section 11.1.1 is implemented for BlobSeer, as presented in Figure 11.1. For simplification purposes, we assume that the only components of BlobSeer that can cause bottlenecks are the data providers. This is a reasonable assumption, as most of the I/O load falls on the data providers.

11.1.3.1 Monitor the data providers

We periodically collect a wide range of parameters that describe the state of each data provider of the BlobSeer instance. For this task we use GMonE [138], a monitoring framework for large-scale distributed systems based on the *publish-subscribe* paradigm. GMonE runs a process called *resource monitor* on every node to be monitored. Each such node publishes monitoring information to one or more *monitoring archives* at regular time intervals. These monitoring archives act as the subscribers and gather the monitoring information in a database, constructing a historical record of the system's evolution.

The *resource monitors* can be customized with *monitoring plugins*, which can be used to adapt the monitoring process to a specific scenario by selecting relevant monitoring information. We developed a plug-in for BlobSeer that is responsible for monitoring each provider and pushing the following parameters into GMonE: number of read operations, number of write operations, free space available, CPU load and memory usage. These parameters represent the state of the provider at any specific moment in time. Every node running a data provider publishes this information every 45 seconds to a single central *monitoring archive* that stores the monitoring information for the whole experiment.

Once the monitoring information is gathered, an aggregation process is undertaken: mean and standard deviation values are calculated for each of the five previous metrics. Additionally, unavailable data providers (due to a failure) are also included as an extra globally

monitored parameter. We call the result of the gathering and aggregation the *global history record* of the behavior of BlobSeer.

11.1.3.2 Identify behavior patterns

The historical data mentioned above is then fed into GloBeM in order to classify the behavior of BlobSeer as a whole into a set of states, each corresponding to a behavior pattern. Thanks to GloBeM, this process is fully automated and we obtain a comprehensive characterization of the states in terms of the most important parameters that contribute to it.

11.1.3.3 Classify behavior patterns according to feedback

For each data transfer, we take the following parameters as client-side quality of service indicators: (i) the effective observed throughput; and (ii) the number of times the operation was interrupted by a fault and had to be restarted. This information is logged for each data transfer, averaged for each state of the behavior model and then used to classify the states into desirable states that offer good performance to the clients and undesirable states that offer poor performance to the clients.

11.1.3.4 Predict and prevent undesired behavior patterns

Finally, the challenge is to improve the behavior of BlobSeer in such way as to avoid undesirable states. This step is completely dependent on the behavioral analysis of the model generated by GloBeM. Since the model is tightly coupled with the data-access pattern of the application, the infrastructure used to deploy BlobSeer, and the corresponding failure model, we detail this step for each of our experiments separately in Section 11.2.3.

11.2 Experimental evaluation

11.2.1 Application scenario: MapReduce data gathering and analysis

We evaluate the effectiveness of our approach for simulated MapReduce workloads that correspond to a typical data-intensive computing scenario on clouds: continuously acquiring (and possibly updating) very large datasets of unstructured data while performing large-scale computations over the data.

For example, a startup might want to invest money into a cloud application that crawls the web in search for new text content (such as web pages) in order to build aggregated statistics and infer new knowledge about a topic of interest.

In this context, simulating the corresponding MapReduce workloads involves two aspects: (i) a write access pattern that corresponds to constant data gathering and maintenance of data in the system and (ii) a read access pattern that corresponds to the data processing. Observe that in most cases the final result of the data processing are small aggregated values that generate negligible writes. Moreover, intermediate data is not persistently stored by MapReduce.

Write access pattern. As explained in [53], managing a very large set of small files is not feasible. Therefore, data is typically gathered in a few files of great size. Moreover, experience with data-intensive applications has shown that these very large files are generated mostly by appending records concurrently, and seldom overwriting any record. To reproduce this behavior in BlobSeer, we create a small number of BLOBs and have a set of clients (corresponding to "map" tasks) generate and write random data concurrently to the BLOBs. Each client predominantly appends, and occasionally overwrites chunks of 64 MB to a randomly selected BLOB at random time intervals, sleeping meanwhile. The frequency of writes corresponds to an overall constant write pressure of 1 MB/s on each of the data provides of BlobSeer throughout the duration of the experiment. It corresponds to the maximal rate that a single web crawler process can achieve under normal circumstances.

Read access pattern. In order to model the data processing aspect, we simulate MapReduce applications that scan the whole dataset in parallel and compute some aggregated statistics about it. This translates into a highly concurrent read access pattern to the same BLOB. We implemented clients that perform parallel reads of chunks of 64MB (which is the default chunk size used in Hadoop MapReduce) from the same BLOB version and then simulate a "map phase" on this data by keeping the CPU busy. Globally, we strive to achieve an average I/O time to computation time ratio of 1:7, which is intended to account for the CPU time of both the "map phase" and the "reduce phase".

We execute both the data gathering and data processing concurrently in order to simulate a realistic setting where data is constantly analyzed while updates are processed in the background. We implemented the clients in such a way as to target an overall write to read ratio of 1:10. This comes from the fact that in practice multiple MapReduce passes over the same data are necessary to achieve the final result.

11.2.2 Experimental setup

We performed our experiments on Grid'5000, using 130 of the nodes of Lille cluster and 275 of the nodes of Orsay cluster. Since MapReduce-style computing systems are traditionally running on commodity hardware, collocating computation and storage on the same physical box is common. However, recent proposals advocate the use of converged networks to decouple the computation from storage in order to enable a more flexible and efficient datacenter design [142]. Since both approaches are used by cloud providers, we evaluate the benefits of applying global behavior modeling to BlobSeer in both scenarios. For this purpose, we use the Lille cluster to model collocation of computation and storage node by co-deploying a client process with a data provider process on the same node, and the Orsay cluster to model decoupled computation and storage by running the client and the data provider on different nodes. In both scenarios, we deploy on each node a GMonE resource monitor that is responsible to collect the monitoring data throughout the experimentation. Further, in each of the clusters we reserve a special node to act as the GMonE monitoring archive that collects the monitoring information from all resource monitors. We will refer from now on to the scenario that models collocation of computation and storage on the Lille cluster simply as *setting A* and to the scenario that models decoupled computation and storage on the Orsay cluster as *setting B*.

Table 11.1: Global states - Setting A

parameter	State 1	State 2	State 3	State 4
Avg. read ops.	68.9	121.2	60.0	98.7
Read ops stdev.	10.5	15.8	9.9	16.7
Avg. write ops.	43.2	38.4	45.3	38.5
Write ops stdev.	4.9	4.7	5.2	7.4
Free space stdev.	3.1*e*7	82.1*e*7	84.6*e*7	89.4*e*7
Nr. of providers	107.0	102.7	96.4	97.2

Since real large-scale distributed environments are subject to failures, we implemented a data provider failure-injection framework that models failure patterns as observed in real large-scale systems build from commodity hardware that run for long periods of time. This framework is then used to simulate failures during run-time, both for setting A and setting B. The failure distribution is generated according to a multi-state resource availability characterization study described in [134].

11.2.3 Results

We perform a multi-stage experimentation that involves: (i) running an original BlobSeer instance under the data-intensive access pattern and failure scenario described; (ii) applying our approach to analyze the behavior of BlobSeer and identify potential improvements; and finally (iii) running an improved BlobSeer instance in the same conditions as the original instance, comparing the results and proving that the improvement hinted by the proposed methodology was indeed successful in raising the performance of BlobSeer. This multi-stage experimentation is performed for both settings A and B described in Section 11.2.

11.2.3.1 Running the original BlobSeer instance

We deploy a BlobSeer instance in both settings A and B and monitor it using GMonE. Both experimental settings have a fixed duration of 10 hours. During the experiments, the data-intensive workload accessed a total of $\simeq$ 11 TB of data on setting A, out of which $\simeq$ 1.3 TB were written and the rest read. Similarly, a total of $\simeq$ 17 TB of data was generated on setting B, out of which $\simeq$ 1.5 TB were written and the rest read.

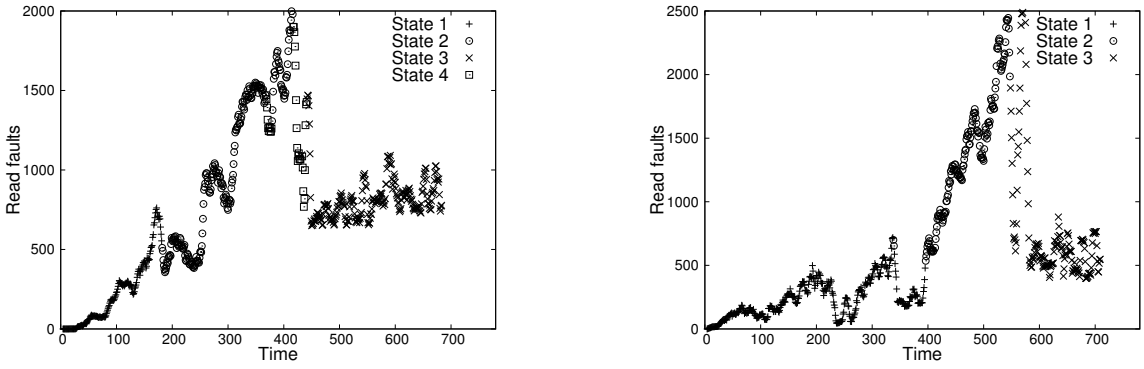
11.2.3.2 Performing the global behavior modeling

We apply GloBeM both for setting A and setting B in order to generate the corresponding global behavior model. Each of the identified states of the model corresponds to a specific behavior pattern and contains the most significant parameters that characterize the state. Tables 11.1 and 11.2 show the average values for the most representative parameters of each state, both for setting A and setting B respectively. It is important to remember that these are not all the parameters that were monitored, but only the ones selected by GloBeM as *the most representative*. As can be seen, GloBeM identified four possible states in the case of setting A and three in the case of setting B.

The client-side feedback is gathered from the client logs as explained in Section 11.1.3. Average read bandwidths for each of the states are represented in Table 11.3 for both settings

Table 11.2: Global states - Setting B

parameter	State 1	State 2	State 3
Avg. read ops.	98.6	202.3	125.5
Read ops stdev.	17.7	27.6	21.9
Avg. write ops.	35.2	27.5	33.1
Write ops stdev.	4.5	3.9	4.5
Free space stdev.	17.2*e*6	13.0*e*6	15.5*e*6
Nr. of providers	129.2	126.2	122.0



(a) Setting A (b) Setting B

Figure 11.2: Read faults: states are represented with different point styles

A and B.

Table 11.3: Average read bandwidth

Scenario	State 1	State 2	State 3	State 4
Setting A	24.2	20.1	31.5	23.9
Setting B	50.7	35.0	47.0	

units are MB/s

Figures 11.2(a) and 11.2(b) depict evolution in time of the total number of read faults as observed by the clients for both scenarios. At this point it is important to remember that these are client-related data and, therefore, neither read bandwidth nor failure information was available to GloBeM when identifying the states. Nevertheless, the different global patterns identified correspond to clearly different behavior in terms of client metrics, as shown in Table 11.3 and Figures 11.2(a) and 11.2(b).

As previously described, the GloBeM analysis generated two global behavior models, respectively corresponding to the behavior of BlobSeer in settings A and B. We performed further analysis using the effective read bandwidth and the number of read faults as observed from the client point of view, in order to classify the states of the behavior models into *desired* states (where the performance metrics are satisfactory) and undesired states (where the performance metrics can be improved).

In the case of setting A, *State 2* presents the lowest average read bandwidth ($\simeq$ 20 MB/s). It is also the state where most read faults occur, and where the failure pattern is more erratic. A similar situation occurs with setting B. In this case again *State 2* is the one with the lowest average bandwidth ($\simeq$ 35 MB/s) and the most erratic read fault behavior. We classify these

states (*State 2* in both settings A and B) to be *undesired*, because the worst quality of service is observed from the client point of view.

Considering now the global state characterization provided by GloBeM for both scenarios (Tables 11.1 and 11.2), a distinctive pattern can be identified for these *undesired* states: both have clearly the highest average number of read operations and, specifically in the case of setting B, a high standard deviation for the number of read operations. This indicates a state where the data providers are under heavy read load (hence the high average value) and the read operation completion times are fluctuating (hence the high standard deviation).

11.2.3.3 Improving BlobSeer

Now that the cause for fluctuations in the stability of the throughput has been identified, our objective is to improve BlobSeer's quality of service by implementing a mechanism that avoids reaching the *undesired* states described above (*State 2* in both settings). Since the system is under constant write load in all states for both settings A and B (Tables 11.1 and 11.2), we aim at reducing the total I/O pressure on the data providers by avoiding to allocate providers that are under heavy read load for storage of new chunks.

This in turn improves the read throughput but at the cost of a slightly less balanced chunk distribution. This eventually affects the throughput of future read operations on the newly written data. For this reason, avoiding writes on providers with heavy read loads is just an emergency measure to prevent reaching an *undesired* state. During normal functioning with non-critically high read loads, the original load-balancing strategy for writes can be used.

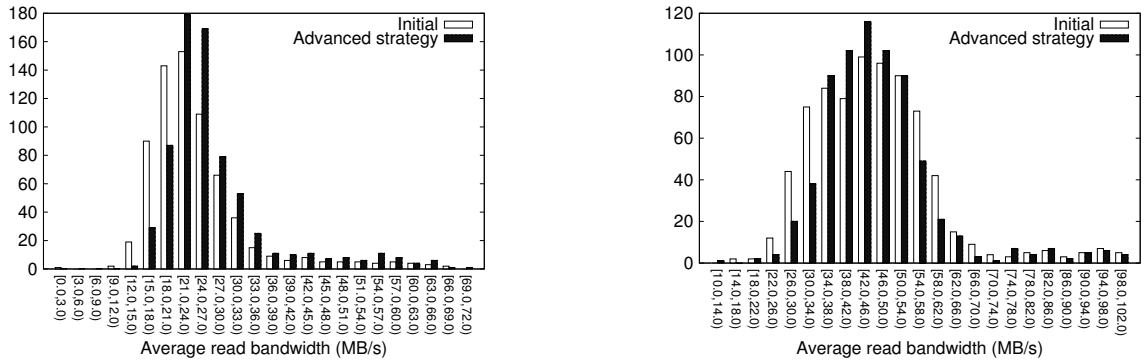
The average read operation characterization provided by GloBeM for *State 2*, which is the *undesired* state (both in settings A and B), is the key threshold to decide when a provider is considered to be under heavy read load and should not store new chunks. We implemented this policy in the chunk allocation strategy of the provider manager. Since data providers report periodically to the provider manager with statistics, we simply avoid selecting providers for which the average number of read operations goes higher than the threshold. We enable choosing those providers again when the number of read operations goes below this threshold.

11.2.3.4 Running the improved BlobSeer instance

The same experiments were again conducted in the exact same conditions, (for both settings A and B), using in this case the improved BlobSeer chunk allocation strategy. As explained, the purpose of this new strategy is to improve the overall quality of service by avoiding the undesirable states identified by GloBeM (*State 2* in both settings A and setting B).

As final measure of the quality of service improvement, a deeper statistical comparison of the average read bandwidth observed by the clients was done. Figures 11.3(a) and 11.3(b) show the read bandwidth distribution for each experimental scenario. In each case, the values of the original and improved BlobSeer version are compared. Additionally, Table 11.4 shows the average and standard deviation observed in each experimental setting.

The results seem to indicate a clear improvement (especially in setting A). However, in order to eliminate the possibility of reaching this conclusion simply because of different biases in the monitoring samples, we need further statistical assessment. In order to declare the



(a) Setting A (b) Setting B

Figure 11.3: Read bandwidth stability: distribution comparison

Table 11.4: Statistical descriptors for read bandwidth (MB/s)

Scenario	mean (MB/s)	standard deviation
Setting A - Initial	24.9	9.6
Setting A - Advanced strategy	27.5	7.3
Setting B - Initial	44.7	10.5
Setting B - Advanced strategy	44.7	8.4

results obtained using the improved BlobSeer implementation (depicted in Figures 11.3(a) and 11.3(b) and Table 11.4) as statistically meaningful with respect to the original implementation, we need to ensure that the different monitoring samples are in fact obtained from different probability distributions. This would certify that the quality of service improvement observed is real, and not a matter of simple bias.

To this end, we ran the Kolmogorov-Smirnov statistical test [149], on both the initial read bandwidth results and the improved read bandwidth results. This test essentially takes two samples as input and outputs a *p-value*, which must be smaller than 0.01 in order to conclude that they *do not originate from the same probability distribution*. The obtained p-values for our samples are with at least one order of magnitude smaller than 0.01.

Finally, the results show a clear quality of service improvement in both settings A and B. In setting A, the average read bandwidth shows a 10% increase and, which is more important, the standard deviation is reduced by almost 25%. This indicates a lesser degree of dispersion in the effective read bandwidth observed, and therefore a much more stable bandwidth (for which the difference between the expected bandwidth (the mean value) and the real bandwidth as measured by the client is lower). Thus, these improvements indicate a significant increase in the overall quality-of-service.

In setting B, the average read bandwidth remained stable, which is understandable given that, as explained in Section 11.2, we are close to the maximum physical hard drive transfer rate limit of the testbed characteristics. Therefore, achieving a higher value is very difficult. Nevertheless, the standard deviation of the read bandwidth was again significantly reduced: no less than 20%.

11.3 Positioning of this contribution with respect to related work

Several approaches to improve quality-of-service have been proposed before. The majority of them relies on modeling.

A large class of modeling approaches, called *white-box*, relies on some foreknowledge of the system and the scenario where it is supposed to run. The most basic approach, benchmarking [41], is insufficient as it relies on manual analysis of monitoring data. Other approaches describe the system formally using Colored Petri Nets (CPN) [19] or Abstract State Machines (ASM) [55] in order to reason about behavior. Rood and Lewis [135] propose a multi-state model and several analysis techniques in order to forecast resource availability.

While these approaches work well for several fixed scenarios, in many cases it is not known how the system looks like and in what context it will be used. To address this need, several so called *black-box* approaches have been proposed. Magpie [14] for example is a tool for automated extraction and analysis of grid workloads. MapReduce frameworks also saw generic diagnosis tools, such as Ganesha [115], that automatically try to identify bottlenecks in arbitrary applications.

Our proposal tries to combine the best of both white-box and black-box approaches. Whereas black-box approaches are more generic, it is their generality that limits their applicability in practice: the modeling process cannot be guided to focus on a specific issue, such as stability of throughput. We use GloBeM to automatically characterize the behavior of the storage service under arbitrary access patterns, but then combine this generic information together with foreknowledge about the implementation of the storage service in order to reason about a specific quality-of-service issue: stability of throughput. This eventually leads to a refined implementation of the storage service. The key in this context is GloBeM, which motivated our modeling choice because of the simple yet comprehensive model it produces as output, which greatly simplifies reasoning about the behavior.

11.4 Conclusions

Global behavioral modeling has been successfully applied to improve the individual throughput stability delivered by BlobSeer, which is an important quality-of-service guarantee that complements the main goal of providing a high aggregated throughput under heavy access concurrency. This brings the potential to use BlobSeer as a cloud storage service that offers advanced features to cloud customers, such as versioning and high throughput under concurrency, while offering a high quality-of-service guarantee: stable throughput for individual data accesses.

Evaluations on the Grid'5000 testbed revealed substantial improvement in individual read throughput stability in the context of MapReduce applications, both for the case when the storage is decoupled from the computation and the case when they share the same machines. More precisely, results show a reduction of standard deviation in read throughput no less than 20% and going as high as 25%. Such numbers are very important to cloud providers, as they can be leveraged to guarantee higher quality-of-service guarantees in the service level agreement, which means more competitive offers that are more appealing to the customers.

Part IV

Conclusions: achievements and perspectives

Chapter 12

Conclusions

Contents

EXISTING approaches to data management in distributed systems face several limitations: poor scalability because of *huge number of small files* and *centralized metadata management*; limited throughput under heavy access concurrency because of *adherence to legacy data access models that were not originally designed for large scales*; *no support for efficient versioning data under concurrency*.

The work presented in this thesis has led to the creation of *BlobSeer*, a highly efficient distributed data storage service that facilitates data sharing at large scale, addressing many of the limitations listed above. We demonstrated the benefits of BlobSeer through extensive experimentations, both in synthetic settings, as well as real-life, applicative settings.

Thus, the main objective proposed in the beginning of this manuscript has been fulfilled. In the rest of this chapter, we underline the usefulness of the contributions presented so far, showing how they enabled us to reach the sub-objectives centered around the main objective. Furthermore, we discuss several choices we made and aspects that were left unexplored and conclude with the potential future perspectives they open.

12.1 Achievements

Key design principles for efficient, large-scale data storage. We have proposed a series of key design principles for building a distributed storage system that can scale to large sizes and can overcome many of the limitations that existing approaches face. First, we advocate for organizing data as *unstructured binary large objects* (BLOBs), which are huge sequences of bytes that aggregate many small application objects. We propose a scheme that

enables fine-grain access to BLOBs under concurrency, eliminating the need to store each application object in a separate file. This avoids having to deal with too many files, which otherwise places a heavy namespace management burden on the storage service. Second, we advocate to enhance an already known principle, *data striping*, to support configurable chunk allocation strategies and dynamically allocatable chunk sizes. These two proposals enable the application to control data-striping at fine-grain level, such that it can potentially exploit the benefits of distributing the I/O workload better than default striping strategies that are uninformed with respect to application intent and access pattern. Third, we propose to *decentralize the metadata management*, which improves scalability and data availability compared to centralized approaches. Finally, we argue in favor of versioning as a crucial principle to enhance parallel access to data and provide support to manage archival data efficiently. We underline the potential of versioning to enable *overlapped data acquisition with data processing* when used explicitly, and to provide a *high-throughput for data accesses under concurrency* when leveraged internally by the storage service. In order to do so, we introduce an asynchronous versioning access interface, and propose the concept of *metadata forward references* respectively. These metadata forward references enhance concurrent access to a degree where synchronization is avoided even at metadata level, greatly reducing the metadata access overhead under concurrency.

BlobSeer: putting these principles into practice. We proposed *BlobSeer*, a distributed data storage service that illustrates the design principles mentioned in the previous paragraph. Our proposal introduces an architecture that is backed up by a series of algorithmic descriptions that formalize our versioning proposal. These algorithms rely on the idea that *data and metadata is added and never removed*, which enables efficient manipulation of BLOBs, providing *total ordering*, *atomicity* and *liveness* guarantees. Furthermore, we introduce a *segment tree based distributed metadata management scheme* that supports metadata forward references and guarantees *logarithmic metadata lookup time for fine grain accesses*. Finally, we detail several important considerations for a practical implementation of these algorithms. Notably, we argue for an *event-driven design* that enables building a highly efficient, asynchronous RPC mechanism; a *DHT-based and consistent hashing based* schemes can be leveraged to implement globally shared data structures that are required by our algorithms in a distributed fashion; a *passive replication scheme* that deals with fault tolerance and data availability without sacrificing performance. We have fully implemented BlobSeer. It is released under the GNU LGPL (http://blobseer.gforge.inria.fr) and is registered with the French software protection agency (APP - Agence pour la Protection des Programmes).

Theoretical benefits demonstrated through synthetic scenarios. We performed extensive synthetic benchmarks using BlobSeer that emphasize the impact of each of the proposed design principles (data striping, distributed metadata management and versioning) on the sustained throughput under concurrent access to the same BLOB. With respect to data striping, our findings show that best results are obtained when at least as many data providers are deployed as clients, as this enables each clients to interact with a potentially different provider and therefore maximizes the distribution of the I/O workload. When this is the case, our approach achieves under concurrency an average throughput per client that is only 12% lower than when no concurrency is present. Moreover, we have also shown that our approach can leverage the networking infrastructure efficiently, achieving a high aggregated throughput

that pushes it to its limits. Finally, we have shown that our approach remains highly scalable even when the clients are co-deployed with data providers, which is a typical case for data-intensive applications that are data location aware. Our distributed metadata management scheme proved to be crucial when the metadata overhead becomes significant due to the large number of chunks involved. In this context, the speedup obtained for concurrent data accesses versus a centralized approach is at least 20%, with several I/O intensive scenarios where the speedup is even more than double. With respect to the benefits of versioning, we show that readers can access a BLOB that is concurrently altered by writers with minimal average throughput loss (and the reverse, too). Our versioning proposal shows high scalability *even for high-performance networking infrastructure* (such as Myrinet), where the maximal average throughput loss under concurrency is less than 20% when the number of concurrent clients doubles.

Real-life benefits demonstrated through applicative scenarios. BlobSeer was successfully applied in three applicative scenarios.

- **A BlobSeer-based storage layer that improves performance of MapReduce applications.** In the context of data-intensive, distributed applications, the performance of data storage and management has a high impact on the total application performance. We designed and implemented a new storage layer for Hadoop, an open-source MapReduce framework. This storage layer, called BlobSeer-based File System (BSFS), significantly improves the sustained throughput in scenarios that exhibit highly concurrent accesses to shared files, compared to Hadoop Distributed File System (HDFS), the default storage layer. We have extensively experimented with synthetic workloads, where BSFS demonstrated high throughput improvements under concurrency, as well as superior scalability and load balancing. Furthermore, we investigated the real-life benefits of BSFS by experimenting with real MapReduce applications from the Yahoo! distribution of Hadoop, where the improvement ranges from 11% to 30%. This work was carried out in collaboration with Diana Moise, Gabriel Antoniu, Luc Bougé and Matthieu Dorier.
- **A BlobSeer-based approach to improve virtual machine image deployment and snapshotting for IaaS clouds.** We have successfully used BlobSeer in the context of cloud computing to improve the management of virtual machine (VM) images by optimizing two key usage patterns: *multi-deployment*, i.e. simultaneous deployment of one or more VM images to multiple nodes and *multi-snapshotting*, i.e. taking a snapshot of multiple VM instances simultaneously. We propose a lazy VM deployment scheme that leverages the efficient fine-grain access provided by BlobSeer, as well as object-versioning to save only local VM image differences back to persistent storage when a snapshot is created, yet provide the illusion that the snapshot is a different, fully independent image. This has an important benefit in that it handles the management of updates independently of the hypervisor, thus greatly improving the portability of VM images, and compensating for the lack of VM image format standardization. We also demonstrate the benefits our approach both through a series of distributed benchmarks and distributed real-life applications that run inside multiple VMs. Our results show important reductions in execution time (as high as 95%), as well as storage space and bandwidth consumption (as high as 90%), when compared to traditional

approaches. Given the pay-as-you-go cloud model, this ultimately brings substantial cost reductions for the end-user. This work was carried out in collaboration with Kate Keahey and John Bresnahan during a 3-month visit at Argonne National Laboratory, USA, as well as Gabriel Antoniu.

A methodology to improve quality-of-service for cloud storage, illustrated on BlobSeer. We proposed and applied a methodology to improve quality-of-service in the context of cloud computing, where BlobSeer acted as a distributed storage service for I/O intensive access patterns generated by MapReduce applications. In this context, there is a need to sustain a stable throughput for each individual access, in addition to achieving a high aggregated throughput under concurrency. We propose a technique that addresses this need based on component monitoring, application-side feedback and behavior pattern analysis to automatically infer useful knowledge about the causes of poor quality of service and provide an easy way to reason about potential improvements. Using a modified allocation strategy that we implemented in BlobSeer, we demonstrated improvements in aggregated throughput in excess of 10%, as well as a substantial reduction of standard deviation for individual access throughputs (as high as 25%). These results enable cloud providers to set higher service level agreement guarantees, which ultimately makes the offer more attractive to customers at the same price. This work was carried out in collaboration with Jesús Montes, María Pérez, Alberto Sánchez and Gabriel Antoniu in the framework of the "SCALing by means of Ubiquitous Storage" (SCALUS) project.

12.2 Perspectives

During the development of this work, several choices were made that shifted the focus towards some directions at the expense of other directions. Although a large array of promising results was obtained so far, there is certainly room for improvement. Thus, in this section we discuss the impact of our choices and how this work can be leveraged to inspire future research directions that address new challenges or aspects insufficiently developed in this work.

Cost-effective storage on clouds. Cloud computing has revolutionized the way we think of acquiring resources by introducing a simple change: allowing users to lease computational resources in a pay-as-you-go fashion. While this is still an emerging technology, it is increasingly adopted in the industry, which ultimately means for the end-user that there will be a highly dynamic selection of offers. In this context, a promising direction to explore is how to build a cloud storage service that tries to minimize the end user costs by leveraging this dynamic market offer to migrate data from one provider to another depending on how the costs fluctuate. For example, a virtualized BlobSeer deployment can be imagined that relies on advanced allocation strategies and replication mechanisms to perform such data migration from one cloud to another transparently, effectively lowering user bills without explicit intervention. Versioning can prove to be highly valuable in this context, as it has the potential to avoid consistency issues related to migration.

Efficient virtual machine storage. In Chapter 10 we presented a virtual machine image cloud repository based on BlobSeer that introduces several techniques to leverage versioning for optimizing deployment and snapshotting of images on many nodes of the cloud simultaneously. While results are very encouraging in this context, we did not integrate our proposal with a real could middleware and used only a simplified service that mimics its functionality. We have plans to integrate our approach with *Nimbus* [71], an open-source toolkit that enables turning a private cluster into an IaaS cloud. The difficulty in this context is how to efficiently leverage the `CLONE` and `COMMIT` primitives at the level of the cloud middleware in such way that they can be both exposed to the user for fine grain control over virtual machine image snapshotting, as well as exploited internally for higher level functionalities, such as checkpointing and migration. Furthermore, in the typical case, the *same initial image* is deployed and snapshotted on many nodes, which means a similar access pattern to the image is expected on the nodes. For example, during the boot process, all nodes read the same parts of the image in the same order. Such similarities are interesting to explore, as they can bring a promising potential for optimizations. For example, additional metadata based on past experience with these access patterns could be stored together with the image in order to build intelligent prefetching strategies.

Versioning in more applicative contexts. We have shown that our approach remains highly scalable when increasing the number of concurrent writers that alter a BLOB while a set of concurrent readers access it, however this advantage was not exploited to its full potential in real life applications. For example, we have developed BSFS as a storage layer for Hadoop MapReduce applications. However, BSFS implements the standard Hadoop MapReduce file access API, which was not designed to take advantage of versioning capabilities or even support for writes at random offsets in the same file (a file can be changed only by appending data). Given the high throughput sustained by BlobSeer under these circumstance, this opens the potential for promising improvements of MapReduce framework implementations, including Hadoop. For example, versioning can be leveraged to optimize more complex MapReduce workflows, in which the output of one MapReduce is the input of another. In many such scenarios, datasets are only locally altered from one MapReduce pass to another: writing parts of the dataset while still being able to access the original dataset (thanks to versioning) could bring significant speedup to the computation and save a lot of temporary storage space. Generalizing the MapReduce example, there are a lot of scientific workflows and data intensive applications where the output of one computation is used as the input of another. As pointed out in several of our publications [108, 109], many times the input and output files are collections of objects and the computations are transformations of these collections that do not necessarily alter every object (for example, the frames of a movie need to pass a set of image processing filters). Using a BLOB to store this collection, versioning can be leveraged to efficiently apply and revert if necessary such transformations, by combining writes and appends with clone and merge.

BlobSeer as a back-end for Petascale computing systems: efficient decoupled I/O and checkpointing support. In the context of the visualization of large-scale scientific simulations, poor concurrency control at the level of I/O frequently leads to situations where the simulation application blocks on output and the visualization application to blocks on input. We propose to adapt concurrency control techniques introduced in BlobSeer in order to op-

timize the level of parallelization between visualization and simulation with respect to I/O to allow periodic data backup and online visualization to proceed without blocking computation and vice-verso. An important observation is the fact that the visualization usually reads data in a different way than it was written by the simulation. This could be leveraged to optimize the data layout for the read-access pattern generated by the visualization. A second research topic relates to the fault tolerance for massively parallel data processing on large-scale platforms. Checkpointing in such a context typically generates a write-intensive, highly-parallel access pattern. As a case study, we focus on MapReduce data processing applications. We will investigate how incremental checkpointing could be used to enhance fault tolerance by efficiently leveraging the principles proposed in BlobSeer.

High data availability. Data availability and fault tolerance are key issues in the design of cloud storage services, as the provider needs to be able to offer strong guarantees to the customers through the service level agreement in order to gain their trust that it is safe to store data remotely on the cloud. To address these issues, we proposed in Section 7.2 a fault tolerance scheme based on passive replication that leverages the fact that BlobSeer keeps data and metadata immutable in order to simplify replica management: replicas can be created in the background without having to worry about replica consistency. This is a powerful idea that was not fully developed in this work. While some synthetic benchmarks were done to evaluate the resilience of data and metadata under faults, these results are still in a preliminary stage and were not included in this work. Some fault tolerance aspects were also tangentially explored in Chapter 11, where the focus is the improvement of quality-of-service. In this context, we adjusted the chunk allocation strategy to adapt to the access pattern better, which ultimately leads to better quality-of-service. Both the the replication mechanism and the allocation strategy are two aspects that are worth to be given closer consideration, as they have a large potential for promising results in the context of data availability.

Security. Security is a crucial feature of any storage service that is intended to facilitate data sharing between untrusted entities. For example, if a storage service is to be exposed in a cloud to the customers, it must implement strong security mechanisms, as the customers cannot be trusted by the cloud provider. While this aspect is beyond the purpose of this work, we acknowledge its importance. Besides traditional authentication and permission management that can be integrated in BlobSeer, there are several aspects specific to its design that can be improved to increase security. For example, in the current design, the client is responsible to build and write the metadata segment tree. However, this step can be safely delegated to trusted entities such as the metadata providers, as all information about the chunks is already available in BlobSeer before building the tree. Such an approach can greatly enhance security, canceling potential attempts at faking metadata by malicious users. Versioning as proposed by BlobSeer is also a source of promising future research on security. For example, one might consider the scenario where the readers and writers do not trust each other, yet they still want to benefit from versioning to share data under concurrency. In this context, new permission models are required that are able to deal with multiple snapshots and the dependencies between them.

References

[1] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. 77, 82

[2] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus striped GridFTP framework and server. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 54, Washington, DC, USA, 2005. IEEE Computer Society. 27

[3] A. L. Ananda, B. H. Tay, and E. K. Koh. A survey of asynchronous remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 26(2):92–109, 1992. 77

[4] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002. 23

[5] Todd Anderson, Yuri Breitbart, Henry F. Korth, and Avishai Wool. Replication, consistency, and practicality: are these mutually exclusive? In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 484–495, New York, NY, USA, 1998. ACM. 47

[6] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000. 11

[7] Gabriel Antoniu, Jean-François Deverge, and Sébastien Monnet. How to bring together fault tolerance and data consistency to enable grid data sharing. *Concurrency and Computation: Practice and Experience*, 18(13):1705–1723, 2006. 28

[8] Henry G. Baker. Iterators: signs of weakness in object-oriented languages. *SIGPLAN OOPS Mess.*, 4(3):18–25, 1993. 77

[9] Mark Baker, Rajkumar Buyya, and Domenico Laforenza. Grids and grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, 32(15):1437–1466, 2002. 25

[10] Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Commun. ACM*, 46(2):43–48, 2003. 79

[11] Roberto Baldoni, Mariangela Contenti, and Antonino Virgillito. The evolution of publish/subscribe communication systems. In *Future Directions in Distributed Computing*, pages 137–141, 2003. 54

[12] Amotz Bar-Noy and Shlomo Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 13–22, New York, NY, USA, 1992. ACM. 115, 122

[13] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag, 1993. 13

[14] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 259–272, 2004. 136

[15] Alessandro Bassi, Micah Beck, Terry Moore, James S. Plank, Martin Swany, Rich Wolski, and Graham Fagg. The Internet Backplane Protocol: a study in resource sharing. *Future Gener. Comput. Syst.*, 19(4):551–562, 2003. 26

[16] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Readings in database systems*, 1:129–139, 1988. 62

[17] Alan Beaulieu. *Learning SQL, Second Edition*. O'Reilly Media, Inc., 2009. 94

[18] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM. 47

[19] Carmen Bratosin, Wil M. P. van der Aalst, Natalia Sidorova, and Nikola Trcka. A reference model for grid architectures and its analysis. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*, pages 898–913, 2008. 136

[20] John Bresnahan, Michael Link, Gaurav Khanna, Zulfikar Imani, Rajkumar Kettimuthu, and Ian Foster. Globus GridFTP: what's new in 2007. In *GridNets '07: Proceedings of the first international conference on Networks for grid applications*, pages 1–5, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 27

[21] Alex Brodsky, Jan Baekgaard Pedersen, and Alan Wagner. On the complexity of buffer allocation in message passing systems. *J. Parallel Distrib. Comput.*, 65(6):692–713, 2005. 79

[22] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series)*. John Wiley & Sons, 2007. 76

[23] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.*, 25(6):599–616, 2009. 17

[24] Mario Cannataro, Domenico Talia, and Pradip K. Srimani. Parallel data intensive computing in scientific and commercial applications. *Parallel Comput.*, 28(5):673–704, 2002. 12

[25] Franck Capello, Thomas Herault, and Jack Dongarra. *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting*. Springer-Verlag, 2007. 13

[26] Franck Cappello, Eddy Caron, Michel Dayde, Frederic Desprez, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, and Olivier Richard. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *Grid '05: Proc. 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 99–106, Seattle, Washington, USA, November 2005. 83

[27] Philip H. Carns, Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. Pvfs: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, 2000. USENIX Association. 24

[28] Henri Casanova. Distributed computing research issues in grid computing. *SIGACT News*, 33(3):50–70, 2002. 14

[29] Damien Cerbelaud, Shishir Garg, and Jeremy Huylebroeck. Opening the clouds: qualitative overview of the state-of-the-art open source VM-based cloud management platforms. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–8, New York, NY, USA, 2009. Springer-Verlag. 19

[30] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distrib. Comput.*, 9(4):173–191, 1996. 41

[31] Antony Chazapis, Athanasia Asiki, Georgios Tsoukalas, Dimitrios Tsoumakos, and Nectarios Koziris. Replica-aware, multi-dimensional range queries in distributed hash tables. *Comput. Commun.*, 33(8):984–996, 2010. 80

[32] Tom Clark. *Designing Storage Area Networks: A Practical Reference for Implementing Storage Area Networks*. Addison-Wesley, 2003. 23

[33] Benoît Claudel, Guillaume Huard, and Olivier Richard. TakTuk, adaptive deployment of remote executions. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100, New York, NY, USA, 2009. ACM. 115, 122

[34] Flaviu Cristian. Basic concepts and issues in fault-tolerant distributed systems. In *Operating Systems of the 90s and Beyond*, pages 119–149, 1991. 81

[35] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *EW '10: Proceedings of the 10th ACM SIGOPS European Workshop*, pages 186–189, New York, NY, USA, 2002. ACM. 76

[36] Wang Dan and Li Maozeng. A range query model based on DHT in P2P system. In *NSWCTC '09: Proceedings of the 2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, pages 670–674, Washington, DC, USA, 2009. IEEE Computer Society. 80

[37] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association. 107

[38] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. 30, 31, 93, 107

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proc. 18th ACM Symposium on Operating System Principles*, 2007. 31, 122

[40] Phillip Dickens and Jeremy Logan. Towards a high performance implementation of MPI-IO on the Lustre file system. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*, pages 870–885, Berlin, Heidelberg, 2008. Springer-Verlag. 24

[41] Jack J. Dongarra and Wolfgang Gentzsch, editors. *Computer benchmarks*. Elsevier, 1993. 136

[42] Heiko Eissfeldt. POSIX: a developer's view of standards. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 24–24, Berkeley, CA, USA, 1997. USENIX Association. 24

[43] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. 54

[44] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002. 32

[45] Brad Fitzpatrick. Distributed caching with memcached. volume 2004, page 5, Seattle, WA, USA, 2004. Specialized Systems Consultants, Inc. 79

[46] Ian Foster. What is the grid? - a three point checklist. *GRIDtoday*, 1(6), July 2002. 14

[47] Ian Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779*, pages 2–13, 2005. 16

[48] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999. 14

[49] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001. 14, 15, 25

[50] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev.*, 31(5):78–91, 1997. 44

[51] Marcel Gagné. Cooking with Linux: still searching for the ultimate linux distro? *Linux J.*, 2007(161):9, 2007. 110

[52] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. The diverse and exploding digital universe: An updated forecast of worldwide information growth through 2011. *IDC*, 2007. 11

[53] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS - Operating Systems Review*, 37(5):29–43, 2003. 30, 32, 38, 39, 95, 131

[54] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992. 62

[55] Yuri Gurevich. Evolving Algebras: An Attempt to Discover Semantics. *EATCS Bulletin*, 43:264–284, February 1991. 136

[56] Ibrahim F. Haddad and Evangeline Paquin. MOSIX: A cluster load-balancing solution for Linux. *Linux J.*, page 6. 13

[57] Jacob Gorm Hansen and Eric Jul. Lithium: virtual machine storage for the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, pages 15–26, New York, NY, USA, 2010. ACM. 122

[58] Val Henson, Matt Ahrens, and Jeff Bonwick. Automatic performance tuning in the Zettabyte File System. In *FAST'03: First USENIX Conference on File and Storage Technologies*, 2003. 30, 62

[59] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. 46

[60] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the Emulab network testbed. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, Berkeley, CA, USA, 2008. USENIX Association. 122

[61] Mike Hibler, Leigh Stoller, Jay Lepreau, Robert Ricci, and Chad Barb. Fast, scalable disk imaging with Frisbee. In *Proc. of the USENIX Annual Technical Conference*, pages 283–296, San Antonio, TX, 2003. 122

[62] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association. 62

[63] Jeffrey K. Hollingsworth and Ethan L. Miller. Using content-derived names for configuration management. In *SSR '97: Proceedings of the 1997 symposium on Software reusability*, pages 104–109, New York, NY, USA, 1997. ACM. 122

[64] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture—a case for object-based file systems in grids. *Concurr. Comput. : Pract. Exper.*, 20(17):2049–2060, 2008. 28

[65] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming*. Addison-Wesley, 2003. 77

[66] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, 2007. 93

[67] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, 1994. 80

[68] Yvon Jégou, Stephane Lantéri, Julien Leduc, Melab Noredine, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006. 83, 97, 114

[69] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM. 79

[70] David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. In *WWW '99: Proceedings of the eighth international conference on World Wide Web*, pages 1203–1213, New York, NY, USA, 1999. Elsevier North-Holland, Inc. 79

[71] Kate Keahey and Tim Freeman. Science clouds: Early experiences in cloud computing for scientific applications. In *CCA '08: Cloud Computing and Its Applications*, Chicago, IL, USA, 2008. 19, 122, 143

[72] J. Kohl and C. Neuman. The Kerberos network authentication service (v5), 1993. 26

[73] Christopher M. Kohlhoff. ASIO: ASynchronous Input/Output. http://think-async.com/Asio, 2010. 77

[74] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM. 122

[75] Ralf Lämmel. Google's MapReduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007. 94

[76] Leslie Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, pages 32–4, 2001. 82

[77] Leslie Lamport and Keith Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998. 82

[78] Erwin Laure and Bob Jones. Enabling grids for e-science: The EGEE project. Technical Report EGEE-PUB-2009-001. 1, Sep 2008. 16

[79] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society. 18

[80] Anthony Liguori and Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *WIOV '08: First Workshop on I/O Virtualization*, 2008. Online Proceedings. 122

[81] Maik A. Lindner, Luis M. Vaquero, Luis Rodero-Merino, and Juan Caceres. Cloud economics: dynamic business models for business on demand. *Int. J. Bus. Inf. Syst.*, 5(4):373–392, 2010. 17

[82] Xingang Liu, Jinpeng Huai, Qin Li, and Tianyu Wo. Network state consistency of virtual machine in live migration. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 727–728, New York, NY, USA, 2010. ACM. 107

[83] Jon Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly, 2009. 29

[84] David Lomet. The evolution of effective B-tree: page organization and techniques: a personal account. *SIGMOD Rec.*, 30(3):64–69, 2001. 62

[85] David Lomet and Betty Salzberg. Access method concurrency with recovery. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 351–360, New York, NY, USA, 1992. ACM. 62

[86] R. Lottiaux, P. Gallard, G. Vallee, C. Morin, and B. Boissinot. OpenMosix, OpenSSI and Kerrighed: a comparative study. In *CCGRID '05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2*, pages 1016–1023, Washington, DC, USA, 2005. IEEE Computer Society. 13

[87] Konstantinos Magoutis. *Exploiting direct-access networking in network-attached storage systems*. PhD thesis, Cambridge, MA, USA, 2003. Adviser-Seltzer, Margo. 23

[88] Cecchi Marco, Capannini Fabio, Dorigo Alvise, Ghiselli Antonia, Giacomini Francesco, Maraschini Alessandro, Marzolla Moreno, Monforte Salvatore, Pacini Fabrizio, Petronzio Luca, and Prelz Francesco. The gLite workload management system. In *GPC '09: Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, pages 256–268, Berlin, Heidelberg, 2009. Springer-Verlag. 16

[89] John Markoff and Saul Hansell. Hiding in plain sight, Google seeks more power. 13

[90] Ben Martin. Using Bonnie++ for filesystem performance benchmarking. *Linux.com*, Online edition, 2008. 118

[91] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984. 62

[92] John M. McQuillan and David C. Walden. Some considerations for a high performance message-based interprocess communication system. *SIGOPS Oper. Syst. Rev.*, 9(3):77–86, 1975. 79

[93] Ralph D. Meeker. Comparative system performance for a Beowulf cluster. *J. Comput. Small Coll.*, 21(1):114–119, 2005. 13

[94] Marsha Meredith, Teresa Carrigan, James Brockman, Timothy Cloninger, Jaroslav Privoznik, and Jeffery Williams. Exploring Beowulf clusters. *J. Comput. Small Coll.*, 18(4):268–284, 2003. 13

[95] Mike Mesnier, Gregory R. Ganger, and Erik Riedel. Object-based storage. *IEEE Communications Magazine*, 41(8):84–90, 2003. 25

[96] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: virtual disks for virtual machines. *SIGOPS Oper. Syst. Rev.*, 42(4):41–54, 2008. 122

[97] C. Mohan and Frank Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *SIGMOD Rec.*, 21(2):371–380, 1992. 62

[98] Jesús Montes, Bogdan Nicolae, Gabriel Antoniu, Alberto Sánchez, and María Pérez. Using global behavior modeling to improve qos in cloud data storage services. In *CloudCom '10: Proc. 2nd IEEE International Conference on Cloud Computing Technology and Science*, Indianapolis, USA, 2010. In press. 4, 126

[99] Jesús Montes, Alberto Sánchez, Julio J. Valdés, María S. Pérez, and Pilar Herrero. The grid as a single entity: Towards a behavior model of the whole grid. In *OTM '08: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part I on On the Move to Meaningful Internet Systems*, pages 886–897, 2008. 128

[100] Jesús Montes, Alberto Sánchez, Julio J. Valdés, María S. Pérez, and Pilar Herrero. Finding order in chaos: a behavior model of the whole grid. *Concurrency and Computation: Practice and Experience*, 22:1386–1415, 2009. 128

[101] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 19–24, New York, NY, USA, 2009. ACM. 19, 122

[102] C. Morin, R. Lottiaux, G. Vallee, P. Gallard, D. Margery, J.-Y. Berthou, and I. D. Scherson. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 277–286, Washington, DC, USA, 2004. IEEE Computer Society. 13

[103] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proc. of the USENIX Annual Technical Conference*, pages 363–378, 2006. 122

[104] Bogdan Nicolae. Blobseer: Efficient data management for data-intensive applications distributed at large-scale. In *IPDPS '10: Proc. 24th IEEE International Symposium on Parallel and Distributed Processing: Workshops and Phd Forum*, pages 1–4, Atlanta, USA, 2010. Best Poster Award. 3, 4

[105] Bogdan Nicolae. High throughput data-compression for cloud storage. In *Globe '10: Proc. 3rd International Conference on Data Management in Grid and P2P Systems*, pages 1–12, Bilbao, Spain, 2010. 4

[106] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Distributed management of massive data: An efficient fine-grain data access scheme. In *VECPAR '08: Proc. 8th International Meeting on High Performance Computing for Computational Science*, pages 532–543, Toulouse, France, 2008. 3, 4

[107] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling lock-free concurrent fine-grain access to massive distributed data: Application to supernovae detection. In *Cluster '08: Proc. IEEE International Conference on Cluster Computing: Poster Session*, pages 310–315, Tsukuba, Japan, 2008. 3, 4

[108] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Blobseer: How to enable efficient versioning for large object storage under heavy access concurrency. In *EDBT/ICDT '09 Workshops*, pages 18–25, Saint-Petersburg, Russia, 2009. ACM. 3, 4, 143

[109] Bogdan Nicolae, Gabriel Antoniu, and Luc Bougé. Enabling high data throughput in desktop grids through decentralized data and metadata management: The blobseer approach. In *Euro-Par '09: Proc. 15th International Euro-Par Conference on Parallel Processing*, pages 404–416, Delft, The Netherlands, 2009. 3, 4, 143

[110] Bogdan Nicolae, Gabriel Antoniu, Luc Bougé, Diana Moise, and Alexandra Carpen-Amarie. Blobseer: Next generation data management for large scale infrastructures. *Journal of Parallel and Distributed Computing*, 2010. In press. 3, 4

[111] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. Going back and forth: Efficient vm image deployment and snapshotting. Research report, INRIA, 2010. RR-7482. 3, 4, 106

[112] Bogdan Nicolae, Diana Moise, Gabriel Antoniu, Luc Bougé, and Matthieu Dorier. Blobseer: Bringing high throughput under heavy concurrency to hadoop map/reduce applications. In *IPDPS '10: Proc. 24th IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, Atlanta, USA, 2010. 3, 4, 94

[113] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proc. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Los Alamitos, CA, USA, 2009. IEEE Computer Society. 20

[114] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM. 94

[115] Xinghao Pan, Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. Ganesha: Black-box diagnosis for MapReduce systems. In *Proceedings of the Second Workshop on Hot Topics in Measurement & Modeling of Computer Systems*, Seattle, WA, USA, 2009. 136

[116] David A. Patterson. Technical perspective: the data center is the computer. *Commun. ACM*, 51(1):105–105, 2008. 93

[117] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM. 94

[118] George C. Philip. Software design guidelines for event-driven programming. *J. Syst. Softw.*, 41(2):79–91, 1998. 76

[119] Jayaprakash Pisharath. *Design and optimization of architectures for data intensive computing*. PhD thesis, Evanston, IL, USA, 2005. Adviser-Choudhary, Alok N. 12

[120] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55, Catania, Sicily, Italy, 2003. Springer. 47

[121] Konstantinos Psounis and Balaji Prabhakar. Efficient randomized Web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Trans. Netw.*, 10(4):441–455, 2002. 79

[122] Sean Quinlan. A cached worm file system. *Softw. Pract. Exper.*, 21(12):1289–1299, 1991. 29

[123] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association. 29, 122

[124] Ioan Raicu. *Many-task computing: bridging the gap between high-throughput computing and high-performance computing*. PhD thesis, Chicago, IL, USA, 2009. Adviser-Foster, Ian. 12

[125] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978. 80

[126] Tejaswi Redkar. *Windows Azure Platform*. Apress, 2010. 19, 32

[127] Darrell Reimer, Arun Thomas, Glenn Ammons, Todd Mummert, Bowen Alpern, and Vasanth Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 111–120, New York, NY, USA, 2008. ACM. 110

[128] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 143–156, Berkeley, CA, USA, 2008. USENIX Association. 122

[129] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: a public DHT service and its uses. In *Proc. 2005 Conf. Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '05)*, pages 73–84, New York, NY, USA, 2005. ACM. 79

[130] Donald Robinson. *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster*. Emereo Pty Ltd, 2008. 18, 19, 20, 31, 107, 122

[131] Ohad Rodeh. B-trees, shadowing, and clones. *Trans. Storage*, 3(4):1–27, 2008. 62

[132] A. Rodriguez, J. Carretero, B. Bergua, and F. Garcia. Resource selection for fast large-scale virtual appliances propagation. In *ISCC 2009: Proc. 2009 IEEE Symposium on Computers and Communications*, pages 824–829, 5-8 2009. 122

[133] Mathilde Romberg. The UNICORE grid infrastructure. *Sci. Program.*, 10(2):149–157, 2002. 16

[134] Brent Rood and Michael J. Lewis. Multi-state grid resource availability characterization. In *GRID '07: Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, pages 42–49, Washington, DC, USA, 2007. IEEE Computer Society. 132

[135] Brent Rood and Michael J. Lewis. Resource availability prediction for improved grid scheduling. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience (e-Science 2008)*, pages 711–718, Washington, DC, USA, 2008. IEEE Computer Society. 136

[136] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992. 122

[137] Bayer Rudolf and McCreight Edward. Organization and maintenance of large ordered indexes. *Software pioneers: contributions to software engineering*, pages 245–262, 2002. 62

[138] A. Sánchez. *Autonomic high performance storage for grid environments based on long term prediction*. PhD thesis, Universidad Politécnica de Madrid, 2008. 129

[139] Dan Sanderson. *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly Media, Inc., 2009. 19

[140] Frank B. Schmuck and Roger L. Haskin. GPFS: A shared-disk file system for large computing clusters. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 231–244. USENIX Association, 2002. 25

[141] Vitaly Semenov. *Complex Systems Concurrent Engineering Collaboration, Technology Innovation and Sustainability*, chapter Semantics-based Reconciliation of Divergent Replicas in Advanced Concurrent Engineering Environments, pages 557–564. Springer Verlag, 2007. 47

[142] Jeffrey Shafer, Scott Rixner, and Alan L. Cox. Datacenter storage architecture for mapreduce applications. In *ACLD: Workshop on Architectural Concerns in Large Datacenters*, 2009. 131

[143] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, 2003. 23

[144] K. Shvachko, H. Huang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, May 2010. 31, 39, 95

[145] Konstantin Shvachko. HDFS scalability: The limits to growth. *;login: - The USENIX Magazine*, 35(2), 2010. 33

[146] B. Silvestre, S. Rossetto, N. Rodriguez, and J.-P. Briot. Flexibility and coordination in event-based, loosely coupled, distributed systems. *Comput. Lang. Syst. Struct.*, 36(2):142–157, 2010. 77

[147] Patrícia Gomes Soares. On remote procedure call. In *CASCON '92: Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research*, pages 215–267. IBM Press, 1992. 50

[148] Guy L Steele. Lambda: The ultimate declarative. Technical report, Cambridge, MA, USA, 1976. 77

[149] M. A. Stephens. EDF statistics for goodness of fit and some comparisons. *Journal of the American Statistical Association*, 69(347):730–737, 1974. 135

[150] Hal Stern. *Managing NFS and NIS*. O'Reilly, 2001. 23

[151] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, Frans F. Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, February 2003. 79

[152] Scott D. Stoller and Fred B. Schneider. Automated analysis of fault-tolerance in distributed systems. *Form. Methods Syst. Des.*, 26(2):183–196, 2005. 80

[153] Michael Stonebraker, Daniel Abadi, David J. DeWitt, Sam Madden, Erik Paulson, Andrew Pavlo, and Alexander Rasin. Mapreduce and parallel DBMSs: friends or foes? *Commun. ACM*, 53(1):64–71, 2010. 94

[154] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1):127–134, 2008. 106

[155] Osamu Tatebe, Kohei Hiraga, and Noriyuki Soda. Gfarm grid file system. *New Generation Computing*, 28:257–275, 2010. 27

[156] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, 1990. 50

[157] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience: Research articles. *Concurr. Comput.: Pract. Exper.*, 17(2-4):323–356, 2005. 13

[158] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. In *SOSP '85: Proceedings of the tenth ACM symposium on Operating systems principles*, pages 2–12, New York, NY, USA, 1985. ACM. 109

[159] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a MapReduce framework. In *Proceedings of the 35th conference on Very Large Databases (VLDB '09)*, pages 1626–1629, 2009. 94

[160] Viet-Trung Tran, Gabriel Antoniu, Bogdan Nicolae, and Luc Bougé. Towards a grid file system based on a large-scale blob management service. In *Grids, P2P and Service Computing*, pages 7–19, Delft, The Netherlands, 2009. 3, 4

[161] J. J. Valdés. Similarity-based heterogeneous neurons in the context of general observational models. *Neural Network World*, 12:499–508, 2002. 128

[162] J. J. Valdés. Virtual reality representation of information systems and decision rules: an exploratory technique for understanding data and knowledge structure. In *RSFDGrC'03: Proceedings of the 9th international conference on Rough sets, fuzzy sets, data mining, and granular computing*, pages 615–618, Berlin, Heidelberg, 2003. Springer Verlag. 128

[163] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009. 17

[164] Jason Venner. *Pro Hadoop*. Apress, 2009. 94

[165] Srikumar Venugopal, Rajkumar Buyya, and Kotagiri Ramamohanarao. A taxonomy of data grids for distributed data sharing, management, and processing. *ACM Comput. Surv.*, 38(1):3, 2006. 25

[166] Cameron G. Walker and Michael J. O'Sullivan. Core-edge design of storage area networks: A single-edge formulation with problem-specific cuts. *Comput. Oper. Res.*, 37(5):916–926, 2010. 23

[167] Sage A. Weil. *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California at Santa Cruz, 2007. Adviser-Brandt, Scott A. 25

[168] Aaron Weiss. Computing in the clouds. *netWorker*, 11(4):16–25, 2007. 17

[169] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009. 18, 31, 94, 95

[170] Garth Gibson Wittawat Tantisiriroj, Swapnil Patil. Data-intensive file systems for internet services: A rose by any other name... Technical Report UCB/EECS-2008-99, Parallel Data Laboratory, October 2008. 96

[171] Hyrum K. Wright and Dewayne E. Perry. Subversion 1.5: A case study in open source release mismanagement. In *FLOSS '09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 13–18, Washington, DC, USA, 2009. IEEE Computer Society. 29

[172] Himanshu Yadava. *The Berkeley DB Book*. Apress, 2007. 79

[173] Jeremy Zawodny. Redis: Lightweight key/value store that goes the extra mile. Linux Magazine, 2009. 79

[174] C. Zheng, G. Shen, S. Li, and S. Shenker. Distributed segment tree: Support range query and cover query over DHT. In *IPTPS '06: Proceedings of the Fifth International Workshop on Peer-to-Peer Systems*, Santa Barbara, California, 2006. 63

[175] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/. 107, 122

[176] File System in UserspacE (FUSE). http://fuse.sourceforge.net. 113

[177] FUSE. http://code.google.com/p/scp-wave/. 122